Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Robótica Educativa y pensamiento computacional

Educational robotics and computational thinking

Cristian Manuel Ángel Díaz

La Laguna, 24 de junio de 2019 Grado en Ingeniería Informática

D. **Eduardo Manuel Segredo González**, con N.I.F. 78.564.242-Z profesor Asociado adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D. **Rafael Arnay del Arco**, con N.I.F. 78.569.591-G profesor Ayudante Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

C E R T I F I C A N

Que la presente memoria titulada:

"Robótica educativa y pensamiento computacional"

ha sido realizada bajo su dirección por D. **Cristian Manuel Ángel Díaz**, con N.I.F. 43.484.608 -A.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 24 de junio de 2019

Agradecimientos

En primer lugar, quiero dar las gracias a mis padres, por haberme permitido con mucho esfuerzo hacerme posible estudiar este grado, y en general a mi familia por estar siempre a mi lado.

En segundo lugar, cada uno de mis compañeros y amigos de la carrera, por tenderme la mano siempre que lo he necesitado, tanto en lo personal como en lo académico, muchas gracias por estos años, ha sido toda una experiencia.

Por último, quisiera agradecer a mis tutores Rafael y Eduardo por proponer este TFG, además de orientarme y ayudarme durante el desarrollo de este.

Licencia

Resumen

El objetivo de este Trabajo de Fin de Grado (TFG) ha sido la creación de una herramienta web libre y gratuita para permitir a cualquier centro educativo que pueda adoptar "Robótica" como asignatura dentro de sus planes de estudios. Dicha herramienta permite diseñar y personalizar un robot haciendo cambios tanto en las dimensiones de este como añadiendo sensores a la carrocería. Tras su creación, se podrá poner a prueba en un entorno de simulación con distintos retos. En dicho entorno podremos definir el comportamiento del robot por medio de programación visual basada en bloques, en los que podremos juntar las instrucciones y la información recogida por los sensores que hayamos añadido a nuestro diseño para superar los desafíos propuestos.

Este proyecto sigue una línea de trabajo ya establecida en el anterior curso académico, ya que la herramienta producto de este TFG será la unificación de tres trabajos, los cuáles solían ser módulos independientes (creación, simulación y definición de comportamiento), que ahora mismo se encuentran completamente integrados dentro de una misma aplicación.

Palabras clave: Robótica Educativa, Pensamiento Computacional, Simulador, Programación Visual, Unity, C#, Blockly, UBlockly, Sistema Educativo.

Abstract

The goal of this final degree project (FFP) has been the creation of a free web tool to allow any school that can adopt "Robotic" as a subject within their curricula. This tool allows to design and customize a robot making changes in both the dimensions of this and adding sensors to the body. After its creation, it can be tested in a simulation environment with different challenges. In this environment, we'll define the behaviour of the robot by means of visual programming based on blocks, in which we will be able to gather the instructions and the information gathered by the sensors that we have added to our design to overcome the challenges proposed.

This project follows a line of work already established in the previous academic year, since the tool product of this FDP will be the unification of three works, which used to be independent modules (creation, simulation and definition of behavior), that are now fully integrated into the same application.

Keywords: Educational Robotics, Computational Thinking, Simulator, Visual programming, Unity, C#, Blockly, UBlockly, Educational System.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1
Introducción

1.1 Motivación

Saber programar hoy en día es una necesidad, ya que todos estamos vinculados con la tecnología sea cual sea nuestra ocupación. Estar formado en esta disciplina permitirá que nos podamos relacionar de forma más natural con la tecnología que nos rodea, además de mejorar la creatividad, la capacidad de pensamiento crítico y estructurado y la habilidad de resolver problemas (**Pensamiento Computacional**) [3][4] sin olvidarnos de que nos puede abrir muchas puertas en el mundo profesional.

Por estos motivos y muchos otros, la enseñanza de este tipo de competencias es más necesaria que nunca, por lo que debemos introducirlas desde edades tempranas para que las nuevas generaciones estén preparadas para la nueva sociedad de las nuevas tecnologías.

La motivación principal del desarrollo de este TFG surge de la posibilidad de poder contribuir a que sea posible incluir asignaturas de programación en los currículos educativos de enseñanza obligatoria al ofrecer un software de simulación libre y gratuito y tan solo necesitar un ordenador con conexión a internet.

En nuestro caso, hemos apostado por la **Robótica Educativa** [1][2][6] para la enseñanza de las competencias anteriormente mencionadas. Este tipo de clases comienzan con la propuesta de un reto que debe ser superado por los alumnos por medio de su ingenio y las herramientas con las cuenten para hacer que el robot supere dicha prueba

Con la herramienta producto de este TFG se podrá crear un robot que podremos personalizar cambiando sus dimensiones y añadiendo sensores, por medio de los cuáles podremos interactuar con el medio y recoger datos, los cuáles nos serán útiles para poder superar los retos propuestos.

La definición del comportamiento que seguirá el robot para superar las pruebas será por medio de programación visual basada en bloques. Podremos usar bloques para definir desde elementos básicos de la programación, como son los bucles o los condicionales hasta el propio repertorio de acciones que puede ejecutar el robot, las cuáles son avanzar, retroceder, girar y detenerse. La unión de todos estos bloques generará un script que permitirá superar el reto a nuestro robot.

1.2 Objetivos

El objetivo de este TFG consiste en la unificación e integración de tres trabajos desarrollados el pasado curso académico en la Universidad de La Laguna, además de conseguir que se pudiera llevar el proyecto a navegadores.

Los tres trabajos en cuestión serían:

- Framework para la creación de un robot modular. **Sergio García de la Iglesia.**
- Simulador de robots para fomentar el pensamiento computacional a través de lenguajes de programación visual. **Eduardo de la Paz González.**
- Robótica Educativa mediante Programación Visual. **Andrea Rodríguez Rivarés.**

Cada uno de estos trabajos constituyen un módulo con una funcionalidad específica, ya que podríamos decir que, a pesar de no estar unidos y tener que comunicarse a través de sistema de paso de ficheros, formaban parte de una misma aplicación (en apartados posteriores se detallan más aspectos acerca de su funcionalidad).

Y relacionado con este objetivo, tenemos otro, el cuál es proporcionar una aplicación gratuita y fácil de incorporar a los centros educativos, es por este motivo que nos hemos decantado por su llevarla a navegadores, y de este modo, permitir que llegue a la mayor cantidad de usuarios posible, ya que no hace falta ninguna pieza física al ser una herramienta completamente hecha a través de software. Todo esto con el objetivo de ayudar a que finalmente se puedan desarrollar competencias relacionadas con el pensamiento computacional contando con el material necesario en los colegios sin barreras económicas de por medio, además de que proyectos así pueden motivar a otros desarrolladores a crear más herramientas que puedan contribuir a este fin.

1.3 Esquema de módulos de la aplicación

A continuación, hablaremos un poco más en detalle de las funcionalidades específicas y de cómo se relacionan entre sí los módulos de la aplicación justo antes de comenzar el desarrollo de este TFG.

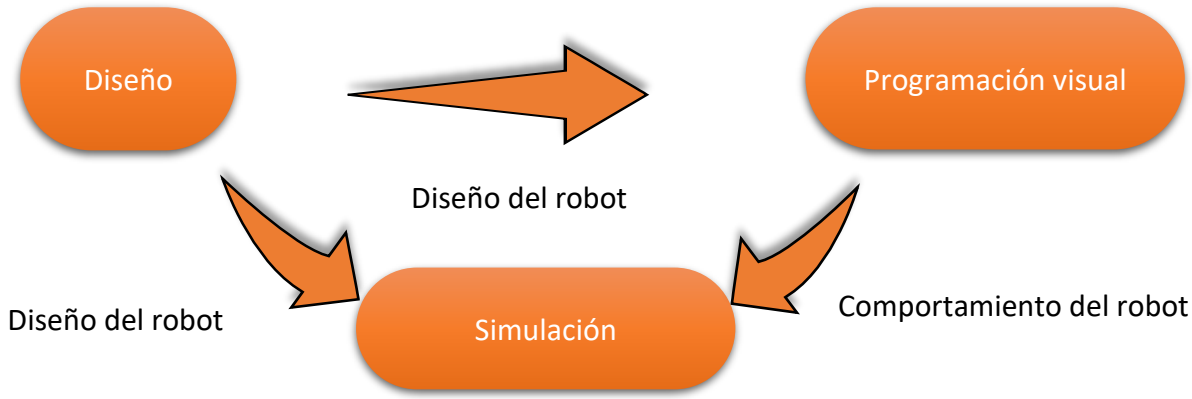

Figura 1: Esquema de relación entre los tres módulos

1.3.1 Diseño del robot

Este módulo permite la creación y diseño de un robot con chasis rectangular, el cuál consta de tres ruedas (dos en la parte anterior y una en la parte posterior). Dicho robot podrá ser modificado a antojo del usuario, como por ejemplo editando el radio de las ruedas, o incluso realizando cambios en las dimensiones del chasis (alto, largo y ancho).

Además de lo mencionado, también tendremos la posibilidad de añadir sensores al robot (**laser, ultrasonido, infrarrojo, de contacto y telémetro láser**), los cuáles podremos localizar en cualquier parte de este. También podremos alterar los parámetros de estos sensores, modificando su radio de acción la precisión que tiene.

Tras terminar nuestro diseño, existe un botón de exportación, con el cuál se generará un fichero XML con la información del robot. Dicho fichero será cargado posteriormente de forma manual por el módulo de simulación y por el módulo de programación visual.

Figura 2: Módulo de diseño

1.3.2 Programación visual del robot

En este segundo módulo consiste en la parte de programar el comportamiento del robot. Para definir este comportamiento se usará programación visual, en concreto se ha usado la librería de Google Blockly [5] para este cometido.

Estos bloques contienen acciones que puede realizar el robot (avanzar, retroceder, girar) y referencias a los sensores que hayamos añadido a nuestro diseño, por medio de los cuáles podremos condicionar las acciones que realizará el autómata en función de la información que devuelvan los sensores del medio.

Una vez se haya finalizado la creación de nuestro código, podremos generar un fichero JavaScript, el cuál contendrá la definición del comportamiento en este lenguaje, con el fin de poder cargarlo más adelante en el módulo de simulación.

Figura 3: Módulo de programación visual

1.3.3 Simulador del robot

Finalmente, en este módulo cargaremos los ficheros creados anteriormente, por una parte, el fichero XML para que se cree el robot con la configuración que ya habíamos definido, por otra parte, el fichero JavaScript con las órdenes que seguirá este. Después de esto solo queda iniciar la simulación, en la que podremos ver cómo se comporta nuestro diseño de robot en un escenario.

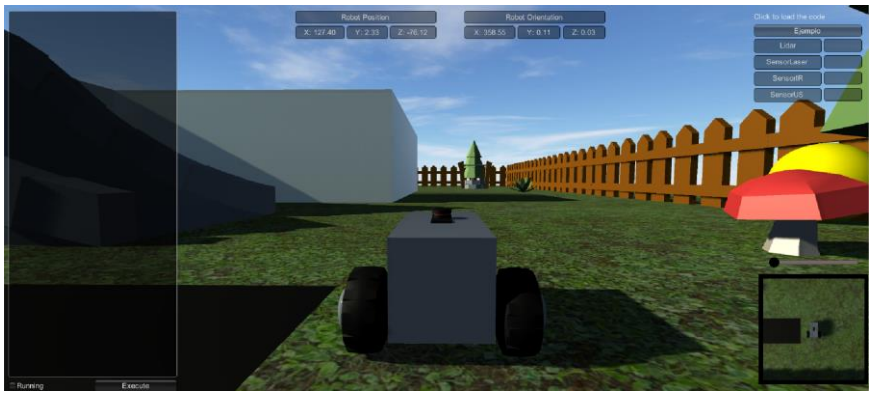

Figura 4: Módulo de simulación

Capítulo 2
Contexto

2.1 Conceptos de interés

2.1.1 Robótica Educativa

La Robótica Educativa es una metodología didáctica que usa a los robots como medio para el impulso de capacidades y competencias a través de la superación de retos, los cuales deberán ser superados por los alumnos haciendo acopio del ingenio y las herramientas que les sean proporcionadas durante esta actividad.

Por lo general, en las clases de Robótica el docente comienza dando unas nociones básicas a los alumnos, para a continuación proponer un problema que debe ser superado. Estas clases contarán con todo el material didáctico necesario para hacer más sencillo el desarrollo de la actividad (además del material para poder desarrollar el robot), como por ejemplo vídeos explicativos, instrucciones o consejos acerca de cómo afrontar el problema.

Podemos separar en distintas fases el desarrollo de una clase de robótica educativa:

1. **Afrontar el problema:** Se trata de investigar y analizar el reto al que nos enfrentamos para comenzar a buscar una solución.
2. **Diseño y creación del robot:** A raíz de lo descubierto en la primera fase de análisis del problema, comenzaremos a diseñar y crear un robot que reúna los requisitos necesarios para superar el reto planteado, teniendo en mente cuál debe ser el comportamiento de este y si tiene los componentes necesarios para poder ejecutar las acciones que le pediremos en el siguiente punto.
3. **Programación de órdenes:** Una vez creado nuestro robot, debemos programar un pequeño script que permita al robot comportarse y moverse de modo que consiga resolver el problema en el entorno de simulación.
4. **Prueba de simulación:** En este punto se deberá probar tanto el diseño como el comportamiento programado y si fuera necesario, se deben hacer cambios tanto en el diseño como en el script. Si el robot supera el reto, habremos encontrado una de las múltiples soluciones a la prueba.
5. **Documentar y presentar:** Se muestran pruebas de que el robot ha superado la prueba de simulación y el alumno comparte el diseño a la clase como alternativa para superar el problema.

Examinando las distintas fases que tiene una clase de robótica, podemos darnos cuenta de que trabajan una gran cantidad de habilidades.

Por un lado, podemos ver las habilidades más técnicas, como son la capacidad de resolución de problemas usando el pensamiento computacional, la lógica y el razonamiento para afrontarlos. Por otra parte, también se fomentan habilidades sociales y comunicativas, como el trabajo en equipo y el habla en público.

A modo de síntesis, podríamos decir que es un sistema de enseñanza que no es convencional, ya que tiene un componente lúdico que son los retos actuando como juegos, esto hace que sea perfecta para introducir a los alumnos a la robótica, y en general a la tecnología.

2.1.2 Pensamiento Computacional

Podemos definir el Pensamiento Computacional como la capacidad de un individuo de afrontar un problema por medio del uso de habilidades normalmente usadas en el área de las ciencias de la computación y del pensamiento crítico.

Tal y como dijo Beatriz Ortega-Ruipérez: "La persona que emplea un pensamiento computacional puede descomponer el problema en pequeños problemas que sean más fáciles de resolver, y reformular cada uno de estos problemas para facilitar su solución por medio de estrategias de resolución de problemas familiares." [7].

Podemos sacar como conclusión que usando este método podemos conseguir resolver problemas complejos de forma más efectiva a una manera habitual.

A continuación, definiremos los pasos usados para la resolución de problemas mediante el Pensamiento Computacional:

- **Descomposición del problema**: Consiste en dividir el problema original en problemas que sean fácilmente solucionables. La solución conjunta de todos estos pequeños problemas conseguirá resolver el problema original.
- **Reconocimiento de patrones:** El siguiente paso es tratar de reconocer similitudes entre los distintos pequeños problemas, los cuáles servirán para atajar la resolución de estos problemas si hay algunos que compartan la misma solución o parte de ella.
- **Abstracción:** Se trata del grado de detalle con el que tratamos el problema analizando sus propiedades por capas ignorando el resto, de modo que podamos destacar sus propiedades específicas, las cuáles las diferencian del conjunto. [8]
- **Diseño algorítmico:** A raíz de las fases anteriores, plantear una serie de pasos (algoritmo) que logren resolver el problema procurando generalizar, de modo que problemas similares puedan ser resueltos con un algoritmo igual o que guarde mucha similitud al planteado.

2.2 Proyectos relacionados

Este proyecto de TFG persigue la creación de una herramienta de software que permita diseñar, programar y simular un robot para servir como introducción a la robótica y a la programación. No obstante, existen otros proyectos ya existentes que tienen un propósito similar al nuestro.

En los siguientes puntos veremos algunos proyectos relacionados con la línea de este TFG.

2.2.1 V-Rep

Es un simulador de robótica cuya interfaz puede recordar a la de Unity. Para el scripting podemos usar varios lenguajes bastante conocidos, entre los que se encuentran C/C++, Python, Java o Matlab.

Nos brinda la posibilidad de implementar de forma rápida nuestros diseños de autómatas y tiene aplicaciones tanto a nivel industrial como a nivel educacional. Cuenta con muchas funcionalidades, como cálculos de cinemática directa/inversa o la capacidad de agregar sensores a nuestro robot para poder interactuar con el entorno. Es compatible con Linux, MacOS y Windows y cuenta con versiones de código abierto y una más avanzada, la cual es gratuita para el uso docente [9][10].

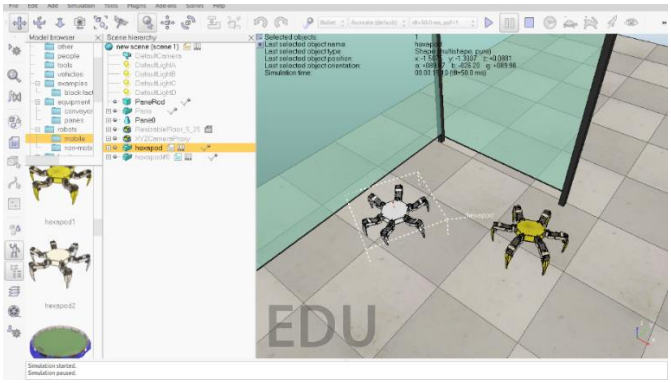

Figura 5: Vrep

2.2.2 CoderZ

Es una solución basada en la nube orientada exclusivamente al ámbito educacional muy completa. Permite usar tanto Blockly para los iniciados como a la codificación en Java para la creación de robots virtuales en 3D cuyo simulador promete ser fiel a la realidad.

Cuenta con un plan de estudios definido y muchas herramientas que ayudan a la enseñanza a través de este entorno, como presentaciones, modelos de evaluación y plantillas que permiten hacer demostraciones. No es una solución gratuita, pero es bastante asequible, lo que permite que muchos centros educativos hagan uso de esta [11][12].

Figura 6: CoderZ

2.2.3 Dash

A priori podría parecer un simple juguete para niños, pero este producto está lleno de posibilidades. Cuenta con una gran cantidad de sensores con los que podrá moverse y sortear obstáculos. Al estar montado y permitir su programación a través de dispositivos móviles con iOS o Android por medio de Blockly, lo hace muy accesible a los niños y permite que los padres aporten su grano de arena al ser muy sencillo de usar y aprender en casa. Es una buena solución para el hogar, a pesar de que el precio sea algo elevado [13].

Figura 7: Dash

Capítulo 3
Metodología

3.1 Fases de desarrollo

La distribución de tareas de la fase de desarrollo fue planteada de la siguiente forma desde la elaboración del anteproyecto de este TFG:

- **Actividad 1:** Buscar la bibliografía necesaria para obtener conocimiento acerca de la Robótica Educacional y la relevancia que puede tener en el sistema educativo, tanto a nivel de oportunidades laborales como el desarrollo de habilidades relacionadas con el pensamiento computacional. A su vez, realizar una búsqueda acerca de trabajos que tengan un propósito similar a la que se pretende desarrollar en este proyecto, lo que ayudará a la hora de plantearnos el diseño del programa o crear mejoras respecto a otros ya existentes.
- **Actividad 2:** Estudiar las herramientas necesarias y familiarizarse con los proyectos a unificar. Antes de comenzar el desarrollo, es necesario conocer Unity y Blockly, ya que ambas fueron los pilares para la creación de los módulos que se tratan de integrar. Además, se deberá investigar cómo funcionan los mismos y cómo se relacionan entre sí para poder vincularlos correctamente.
- **Actividad 3:** Desarrollo e integración completa de los módulos de simulación y creación del robot en Unity. Se pretende unir estos en un único módulo con el fin de hacer más sencilla la labor de exportarlos a código HTML, además de hacer que se comuniquen mejor. A medida que se vayan integrando ambos módulos se deben ir solucionando las posibles incompatibilidades que pueda tener con los navegadores. Se perderá el intérprete de JavaScript con el que cuenta el simulador, ya que no será necesario.
- **Actividad 4:** Integración del módulo unificado de Unity con el módulo de Blockly en el navegador. Al haber eliminado el intérprete de JavaScript del simulador de robots, se pretende que éste acepte órdenes directas del módulo de Blockly en tiempo de ejecución. Se deberá encontrar el intérprete que permita llevar a cabo dicho diálogo haciendo que sea compatible con los navegadores. Una vez hecho esto, se creará una landing page que presentará el proyecto listo para ser usado.
- **Actividad 5:** Elaboración de la memoria de trabajo, la cual se deberá ir rellenando conforme el proyecto vaya evolucionando con la bibliografía de las investigaciones, así como información relativa al desarrollo del proyecto e información de uso.

3.2 Tecnologías y recursos utilizados

3.2.1 Unity

Unity [14][15][16] es un motor gráfico desarrollado por Unity Technologies. Es uno de los más usados a nivel mundial para la creación de contenido interactivo tanto 2D como 3D, generalmente videojuegos. Está disponible para la gran mayoría de sistemas operativos (Windows, MacOS y Linux) y los proyectos pueden ser exportados a una gran lista de plataformas, entre los más destacados podemos nombrar los dispositivos móviles, las videoconsolas de última generación o los principales navegadores de web.

Para este proyecto estamos usando la versión Unity Personal 2019.1.0f2 (64-bit). Aparte de esta versión personal, la cual está disponible de forma gratuita, existen otras dos versiones, la versión Unity Plus, orientada a desarrolladores aficionados que no generen más de 200.000$ en ganancias anuales, y la Unity Pro, para aquellos que superen esta cota de ingresos.

Concretamente para este proyecto estamos usando Unity3D, ya que para la simulación del robot requeríamos de un entorno tridimensional en el que pudiéramos las posibles colisiones que pudiera tener el autómata con los elementos del entorno. Con Unity3D la codificación se puede hacer a través de los lenguajes C# y JavaScript, los cuáles van de la mano con el IDE MonoDevelop, además, cuenta con un editor visual muy potente e intuitivo el cual nos ayudará a agilizar el desarrollo de nuestros proyectos, pudiendo crear con facilidad nuestros propios entornos 3D, o importar recursos creados por otros desarrolladores de la comunidad a través de la Asset Store, que van desde modelos 3D, hasta texturas o efectos de sonido.

Hasta el comienzo del desarrollo de este TFG no había tenido contacto con Unity ni con el lenguaje C#, usado para la codificación, aunque personalmente no me ha supuesto ningún problema gracias a la curva de aprendizaje de esta tecnología y a la gran cantidad de recursos online que podemos encontrar para aprender a usar esta herramienta.

Vamos a pasar a definir algunos de los conceptos más importantes de Unity para entender un poco mejor como funciona este motor gráfico:

- **GameObject:** Es el elemento principal de Unity. Dichos elementos se encuentran contenidos dentro de las escenas. Como tal, el GameObject es un elemento vacío, que de forma nativa solo cuenta con un componente que define su posición, rotación y escala dentro de la escena. Deberemos añadirle componentes a nuestro GameObject para cambiar las propiedades y el comportamiento de este en la escena. Un GameObject podría ser desde una piedra o un árbol, hasta una luz que ilumine los elementos de la escena.
- **Componente:** Son aquellos elementos que al añadirse a un GameObject modifican sus propiedades. Estos son los algunos de los componentes más importantes.
    - **Scripts:** Por medio de la codificación de scripts podremos tanto alterar parámetros de nuestro GameObject como modificar el comportamiento de este en la escena, como por ejemplo haciendo que responda con una determinada acción tras la pulsación de una tecla [17].
    - **Transform:** Es un componente obligatorio para cualquier GameObject. Sirve para localizarlo dentro de la escena, además de para cambiar su rotación y su escala.
    - **Mesh:** Este elemento permite definir la malla de puntos con la que podremos definir cómo será renderizado el GameObject en la escena.
    - **Collider:** Con este componente podremos configurar una malla de colisión, la cuál no tiene por qué seguir la geometría definida en el componente Mesh, por ejemplo, podemos tener un GameObject que se renderice como un cuadrado, pero que su malla de colisiones sea la de una esfera.
    - **Rigidbody:** Al añadir un componente de este tipo a un objeto, estaremos indicando que este se encuentra bajo el control de la física. Esto quiere decir que los GameObjects que tengan dicho componente se verán influenciados por la gravedad y otro tipo de fuerzas que pueden ser tanto fruto de una colisión con un elemento de la escena o por medio de su aplicación vía script [18].
- **Escena:** Contiene los entornos y menús creados a base de GameObjects. Cada escena debe contener al menos una cámara para poder visualizar lo que esté pasando durante la ejecución de esta escena y una luz direccional para iluminar el entorno [20].

- **Prefab:** Tras haber creado y modificado un GameObject tenemos la posibilidad de guardar un modelo de este con todas sus propiedades. Este modelo puede ser usado posteriormente para ser instanciado en las escenas de nuestro proyecto. Además, si modificamos un Prefab, automáticamente todas las instancias que existan de este tendrán las modificaciones que hemos [19].

3.2.2 UBlockly

UBlockly [21] es una reinterpretación de Google Blockly desarrollada con Unity. Es un proyecto desarrollado por el desarrollador Ling Mao que permite llevar de una forma más sencilla y eficiente la programación por bloques de Blockly a Unity.

En un inicio se barajó la posibilidad de usar Google Blockly, ya que el módulo de programación visual del robot había sido desarrollado con esta tecnología, por lo que, tras exportar los módulos de creación y simulación a navegadores (ambos módulos debidamente integrados), la tarea a realizar sería hacer que la comunicación entre los módulos de Unity y el de Blockly se comunicaran de forma más eficiente y en tiempo real.

Finalmente se descartó esta posibilidad, ya que existía una gran lista de problemas e inconvenientes a esto. [22]

- La versión web requiere un complemento de terceros
- No se admiten funciones de Unity como las corrutinas.
- No se puede utilizar UGUI, la flexibilidad de diseño de interacción de la UI es baja.
- El coste de tiempo para realizar esta comunicación es bastante elevado, además, no se garantiza un resultado óptimo, ya que la comunidad reporta unos grandes tiempos de espera entre el envío e interpretación de mensajes.

Por estos motivos decidí continuar investigando para finalmente encontrar este proyecto, el cuál conseguiría ahorrar mucho trabajo y hacer que el proyecto fuera más sólido, ya que todo estaría desarrollado únicamente con Unity, y la comunicación entre el robot y los bloques sería interna, lo cual haría que fuera instantánea.

UBlockly cuenta con tres módulos que se comunican entre sí, estos serían *Code, Model* y *UI.*

- *Code:* Este módulo es el núcleo de funcionamiento de UBlockly. Tras la creación de nuestro script visual, permite **generar** el código C# fruto de la unión de los bloques del área de trabajo, ya que C# es un lenguaje estático que no permite generar código de forma dinámica se necesita **interpretar** el código asociado al bloque como una implementación de C#, esta interpretación será por medio del uso de IEnumerator, ya que consigue que el rendimiento en ejecución no se vea limitado y la conectividad entre bloques requiere de un método que permita la reentrada al código. Por último, tendríamos la parte de **ejecución** que haría que el código empezara a funcionar. Hay algunos pequeños inconvenientes a raíz del uso de IEnumerator, entre ellos que no se puede pausar y reanudar la ejecución, lo cual a nosotros no nos supone ningún problema, ya que solo vamos a necesitar detenerla sin guardar el estado en el que se encontraba la ejecución del código [24].

- *Model:* Constituido por el espacio de trabajo, el cual contiene las variables y los bloques.
    - **Variables**: Son globales a todo el espacio de trabajo.
    - **Bloques**: Representan a un programa ejecutable, los cuáles pueden tener (o no) una salida, de forma análoga a una función en programación tradicional. Un bloque debe

tener, al menos, un tipo de conexión para que se pueda conectar con otro bloque. Estas conexiones pueden ser de entrada, salida, anterior y posterior. Las conexiones entrada – salida permiten que un bloque reciba un parámetro fruto de la ejecución del bloque de salida. Las conexiones anterior – posterior indican cuál es el orden de ejecución de los bloques. Los bloques son definidos en ficheros .json con su estructura y propiedades, como por ejemplo su color, o si requiere que se introduzca un dato de entrada, como un número o texto [23].

- *UI:* Se ocupa de gestionar toda la parte visual con la que interactuará el usuario. Este módulo se encarga de crear los Prefabs de los bloques a partir de la estructura definida en los .json, modificar el tamaño de los bloques en el área de trabajo en tiempo de ejecución si fuera necesario [25].

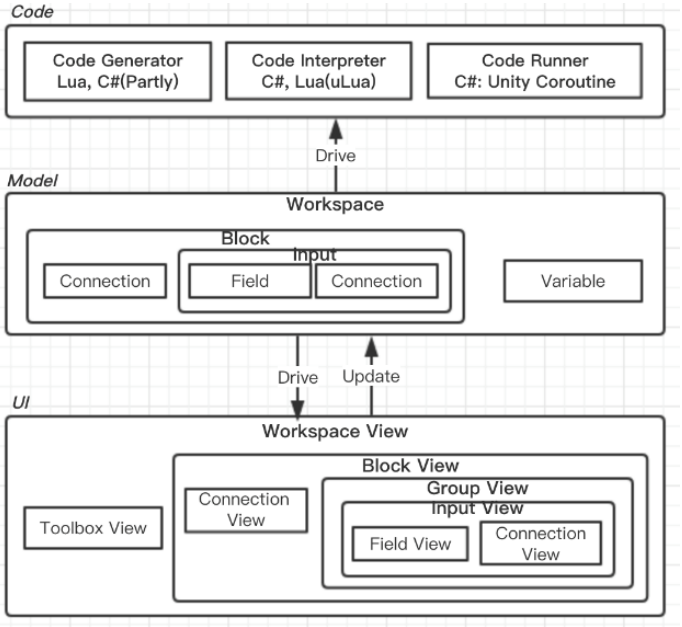

Figura 8: Esquema de funcionamiento de UBlockly

Desgraciadamente más allá de la bibliografía ofrecida al final de este documento no existe más documentación o información acerca de este proyecto y la forma de integrarlo dentro de nuestros proyectos de Unity, lo que ha añadido algo de dificultad a todo este proceso. Aun así, lo consideramos la alternativa más viable, y descubrir la forma de trabajar con él fue cuestión de familiarizarse con el proyecto y leer código.

Para añadir UBlockly a nuestro proyecto el primer paso será introducir el contenido de la librería dentro de la carpeta de nuestro proyecto, concretamente la he ubicado en `Assets/Plugins`.

De serie UBlockly tiene varias categorías de bloques creadas, como los condicionales o los bucles. Para poder aprovecharlo se crearon bloques específicos a las acciones o información que podía recoger el robot del medio con ayuda de los sensores.

```
{
    "type": "move_robot_forward",
    "message0:"%{BKY_MOVE_ROBOT}
               %{BKY_MOVE_ROBOT_FORWARD}
               con velocidad %1",
    "args0": [
        {
            "type": "field_number",
            "name": "DISTANCE",
            "value": 1,
            "min": 1,
            "max": 5,
            "int": true
        }
    ],
    "previousStatement": null,
    "nextStatement": null
},
```

Fichero de definición de bloques

```
"MOVE_ROBOT": "Mover robot",
"MOVE_ROBOT_FORWARD": "hacia delante",
```

Fichero de diccionario

A continuación, se muestra un pequeño ejemplo de cómo se crea un bloque en UBlockly:

- **type**: Apartado en el que nombraremos el nombre del bloque que vayamos a definir.
- **message0**: Constituye el mensaje que verá el usuario escrito dentro del bloque, puede contener menciones a diccionarios, cuya sintaxis es un símbolo de porcentaje "%" junto con BKY_ y el nombre del elemento del diccionario contenido entre llaves "{}". También podemos poner argumentos con un "%" seguido de un número, este número será una referencia al índice del argumento.
- **args**: Sección en la que definiremos los argumentos que tiene nuestro mensaje, existen varios tipos de campos para los distintos tipos de dato, como pueden ser numérico, booleanos, cadenas de caracteres o listas con distintas opciones. A cada argumento se le asigna un nombre y posteriormente las distintas opciones específicas de cada tipo de campo.
- **previousStatement** y **nextStatement**: Dependiendo de si toma o no valores nulos, podemos especificar si el bloque tiene conexiones anterior o posterior.
- **output**: Indicamos que el bloque devuelve un dato, y el tipo de dato.

Tras haber definido los bloques debemos buscar en la barra de herramientas el botón "UBlockly", tras clicarlo nos aparecerá la opción "Build Block Prefabs". Esto nos permite crear los prefabs de los bloques que hemos definido anteriormente.

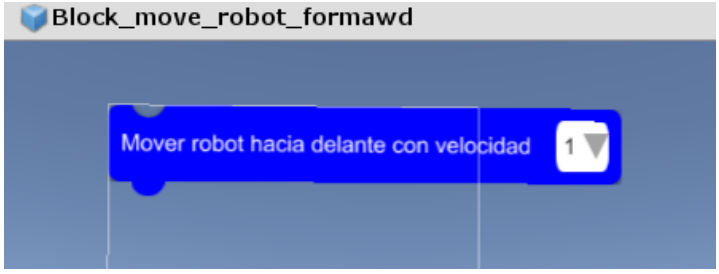

Figura 9: Ejemplo de bloque

3.2.3 Microsoft Office Word

Microsoft Word [26][27] es un software desarrollado por Microsoft Corporation que pertenece a la suite ofimática Microsoft Office. En concreto, Word nos permite crear y editar documentos digitales. Normalmente cada tres o cuatro años suelen sacar una revisión de este software, las cuáles cada vez cuentan con funcionalidades más potentes, ayudando a mejorar la experiencia de usuario y la facilidad a la hora de crear documentos.

Decidí usar esta opción para escribir esta memoria ya que es muy sencilla de usar, y estoy muy habituado a trabajar con ella, ya que la gran mayoría de trabajos y documentos que he hecho a lo largo de mi vida han sido con este software.

Aun así, también me planteé el uso de LaTeX ya que es frecuentemente usado para la elaboración de artículos y documentos con carácter científico, aunque aprender a usar la sintaxis para la creación de documentos me llevaría tiempo adicional para la creación de este escrito.

3.2.4 Módulos de la aplicación

Se han aprovechado algunos de los recursos del proyecto desarrollado el anterior curso académico.

En cuanto al módulo de creación del robot se ha podido aprovechar en su totalidad, eliminando la parte de exportación del fichero XML con la configuración del robot ya que no nos sería de utilidad.

Respecto al módulo de simulación se han aprovechado bastantes partes. Aunque se ha tenido que descartar la funcionalidad de lectura de los ficheros de instrucciones y de diseño del robot ya que no era necesaria. También debemos mencionar que no se ha incluido la librería Jurassic, ya que no necesitamos interpretar código JavaScript, tampoco se ha aprovechado código relacionado con los hilos de ejecución de los bloques, cuya función era conseguir un correcto ciclo de ejecución del código generado por la unión de los bloques, ya que el intérprete de UBlockly soluciona este problema.

Por último, en cuanto al módulo de programación visual no se ha podido aprovechar nada de material, debido a la decisión que se tomó de no continuar usando Google Blockly, aunque se ha tenido en cuenta el diseño de las instrucciones de los bloques.

3.3 Desarrollo del proyecto

3.3.1 Unión de los módulos de Creación y Simulación.

El primer paso en el desarrollo de este TFG fue la unión de estos dos módulos. Al estar ambos desarrollados en Unity eran por así decirlo “compatibles”. El resultado final que se buscaba de dicha integración era que se pudiera crear el robot en la escena de simulación y que pasara directamente a la de simulación sin tener que exportar ni cargar ningún fichero.

Para comenzar se unió el contenido de las carpetas de ambos módulos para contar con los assets, prefabs, escenas, scripts y demás recursos en un solo proyecto de Unity.

Tras hacer esto hubo que corregir algunos problemas con incompatibilidades entre la versión de Unity que estaba usando para el desarrollo de mi TFG y las versiones en las que habían sido desarrollados los proyectos que debía unir. Una de estas incompatibilidades era con TextMeshPro, que es un asset que permite una renderización en alta calidad del texto en las escenas, además de que tiene muchas más opciones que Unity UI Text. La incompatibilidad que surgió fue a causa de

que para disponer este asset los desarrolladores de los módulos tuvieron que añadirlo manualmente al proyecto descargándolo de la Asset Store, ya que estaban usando una versión anterior a la Unity 2018.2, ya que a partir de esta viene integrado de serie sin tener que descargar ningún paquete.

Al descubrir esto me dispuse a eliminar todo rastro del paquete TextMeshPro de las carpetas de assets. A continuación, hubo que actualizar todos los textos que estuvieran en las escenas cambiándolos al TextMeshPro integrado, además de modificar las referencias que había en el código a estos textos, ya que se había cambiado la nomenclatura para nombrar a este tipo de elemento.

El siguiente paso fue conseguir que el GameObject del robot pasara de una escena a otra sin paso de ficheros. Documentándome encontré un método conocido como *DontDestroyOnLoad*, que permitía no destruir el GameObject y su contenido cuando se cambiara de escena. Para conseguir mi objetivo creé dos prefabs, uno del robot estándar añadiéndole todos los scripts que necesita para ambas escenas y otro llamado "Permanente" que contiene un script que contiene un pequeño script que hace que no se borre al pasar de escena. Coloqué el GameObject "Permanente" en la escena de Creación con el GameObject del robot como hijo, y en la escena de Simulación coloqué "Permanente" sin ningún GameObject anidado.

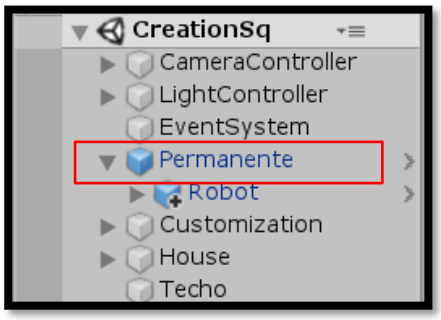

Figura 10: Permanente en la escena de creación

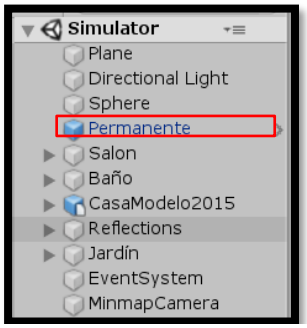

Figura 11: Permanente en la escena de simulación

También se habilitó un botón que permite el cambio de escena. Cuando conseguí que el paso del robot al entorno de simulación funcionara, tuve que llevar a cabo algunas pequeñas modificaciones a nivel de código. Añadí al robot un pequeño script **RobotScriptsController.cs,** el cual hace algunos scripts del robot estén habilitados o deshabilitados dependiendo de la escena que esté activada. Los ficheros que se controlan son **Rotation.cs** que permite al robot girar en la escena de creación usando las flechas y **RobotCollider.cs** que cambia al robot de color en la escena de simulación cuando entra en contacto con una moneda situada en el entorno, además de activar o desactivar la cámara que usa el robot en la escena de simulación.

La activación y desactivación de estos scripts es crucial, ya que permite controlar las funciones que puede hacer el robot y alterar parámetros sus parámetros al cambiar de escena mediante la función Start( ), lanzada la primera vez que se ejecuta solo la primera vez que se habilita el script, y OnEnable( ), ejecutada con cada activación del script. Algunos de los parámetros que se modifican son la posición, rotación y escala del robot. También se hacen ajustes relacionados con las físicas de las ruedas por medio de la propiedad isKinematic de los Rigidbody de esos GameObjects o el tamaño de la cámara.

Figura 12: Botón de comienzo de simulación

Por último, se tuvo que hacer unas correcciones en las colisiones entre el robot y los sensores en la escena de simulación. El problema surgió por el hecho de que parte del sensor se encuentra dentro del modelo 3D del robot, y esto generaba una colisión continua entre ambos objetos. La solución consistió en deshabilitad la malla de colisión en las partes del sensor que colisionan con el robot, lo cual no afecta al funcionamiento del sensor ni a la precisión de la simulación.

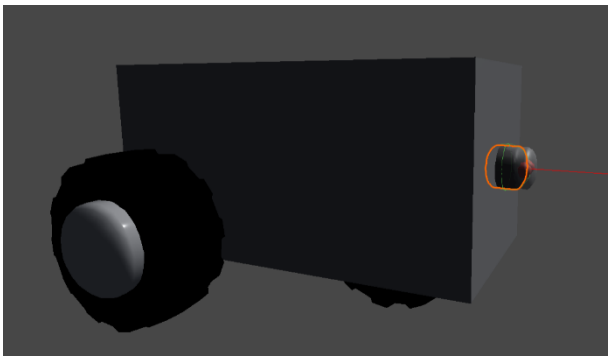

Figura 13: Sensor laser colisionando por la parte trasera con el modelo 3D del robot

El resultado final de la estructuración de carpetas es el siguiente:

- **Carpetas con recursos de la Asset Store heredados del anterior proyecto**:
  - ***AxeyWorks*:** Elementos decorativos de exterior**.**
  - ***BigFurniturePack*:** Elementos decorativos de interior, entre ellos mobiliario.
  - ***Casa*:** Modelo de la casa del entorno de simulación.
  - ***ConeCollider*:** Modelo de colisión de forma cónica usado para el sensor de contacto.
  - ***Ground textures pack y QS*:** Contienen texturas para el suelo, entre ellas una con efecto de césped que es la que se ha aprovechado.
  - ***Wispy Sky*:** Asset que permite generar un cielo más realista en la escena de pruebas del robot.
- **Scenes**: Contiene las distintas escenas con las que cuenta el proyecto.
- **Scripts**: Almacena los scripts necesarios para la creación y simulación del robot. Hay scripts de propósitos varios, aunque buena parte está orientada al control de los sensores, es por este motivo que se decidió crear un subdirectorio que los separe del resto.
- **Prefabs**: Almacena los prefabs de todos los elementos que componen al robot (chasis y

ruedas), además de los de los distintos sensores que podemos acoplar a este.

- **Images**: Contiene las imágenes que se usan en el proyecto.
- **Resources:** En su interior hay una colección de los recursos, tanto texturas como prefabs más usados provenientes de la Asset Store.

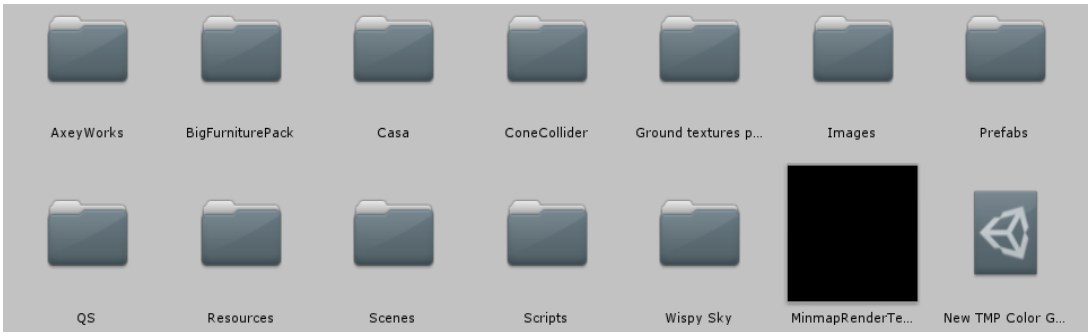

Figura 14: Estructura de directorios conjunta de los módulos de simulación y creación

3.3.2 Unión de los módulos de Unity con el módulo de programación visual.

El siguiente paso tras haber logrado unir con éxito los módulos de Unity era integrar de algún modo el módulo de programación visual. Como se explicó en el punto 3.3.2 de esta memoria, en un principio se pretendía comunicar el módulo de programación visual desarrollado en el anterior curso académico con los módulos de Unity en el navegador, pero esta opción contaba con bastantes inconvenientes, por lo que se decidió descartarla. Tras realizar una tarea de investigación encontré un proyecto conocido como UBlockly, que permitía integrar Blockly directamente en Unity.

Lo primero que hice fue probar unas demos que el desarrollador de este proyecto había dejado para ver su funcionamiento. Al probarlo descubrí que era justo lo que necesitaba, así que me decidí a integrar UBlockly dentro de mi proyecto. Este proceso es bastante sencillo , aunque tuve algunas dificultades para saber cómo llevarlo a cabo, ya que la documentación de esta herramienta no es tan detallada como debería, así que con ayuda de las demos que había descargado, examinando la estructura de carpetas y ficheros logré descubrir cómo hacerlo.

Para comenzar con el proceso de integración, descargué la librería de UBlockly desde GitHub, para a continuación añadir los ficheros al proyecto dando como resultado esta estructura:

- **AllRes/Robot**: Contiene tres subdirectorios que almacenan los ficheros .json con los cuáles se definirán los bloques que se usarán, el fichero de diccionario y la estructura que va a tener la caja de herramientas (toolbox) del área de trabajo de Blockly.
- **Plugins:** Almacena parte del contenido del repositorio de GitHub de UBlockly, concretamente fotografías de la documentación y una estructura simple para una toolbox que solo contiene las categorías básicas junto con la definición de los bloques de cada categoría (una categoría es un conjunto de bloques que comparten una temática en su funcionalidad). También contiene los dos tipos de diseño de toolbox disponibles, que son *ClassicToolbox* (la elegida para este proyecto) y *ScratchToolbox*.
- **ProjectData**: Al ejecutar la orden de creación de los prefabs de los bloques después de haberlos definido en los ficheros json, se almacenan en esta carpeta junto con el resto de prefabs del proyecto. No debemos olvidarnos de mencionar que también contiene dos ficheros muy importantes para la personalización de los bloques y elementos que utilizaremos en la toolbox

del entorno de simulación, además de configurar cómo los visualizaremos, estos dos ficheros son *BlockResSettings* y *BlockViewSettings*.

- **Scripts/UToolBox:** Los scripts de esta carpeta permiten, almacenar y cargar scripts de bloques hechos por el usuario además de limpiar correctamente los objetos al finalizar.
- **Scripts/Common:** Estos scripts permiten la ejecución ordenada y correcta de todas y cada una de las órdenes de los bloques con mecanismos de espera de finalización de frame o espera de condiciones.
- **Scripts/Robot:** En esta carpeta están los ficheros que permiten asignar un fragmento de código a los bloques que hemos definido.
- **Source:** Contiene los intérpretes, generadores y lanzadores de código, así como las definiciones de los distintos cuadros de diálogo, gestores de las conexiones entre bloques y otros ficheros encargados del funcionamiento, gestión y correcta visualización del módulo de programación por bloques.

Figura 15: Estructura de directorios con UBlockly integrado

Una vez hecho esto, creé el área de trabajo dentro de la escena de simulación a partir de un *canvas* que contiene un *prefab* de la librería llamado "Workspace". En ella podemos ver que existe una zona en la que se pueden ver las listas de colecciones de bloques, el área de codificación y algunos botones que nos permiten ejecutar, guardar y cargar nuestros scripts hecho con bloques.

Para poder visualizar tanto el Workspace como el entorno de simulación se llevaron a cabo algunos ajustes en la cámara. En concreto el GameObject "Robot Camera", que es la cámara que apunta al robot cambia sus valores al cambiar a la escena de simulación en el parámetro "Rect", de modo que solo se visualizará el robot en una esquina para poder trabajar cómodamente con el espacio de trabajo de UBlockly. Este sería el fragmento de código del fichero "RobotCollider.cs" que se activa al cambiar a la escena de simulación.

```
void Start(){
    main = GameObject.Find("Robot Camera").GetComponent<Camera>();
    main.rect = new Rect(0.65f,-0.45f,1,0.8f);
}
```

Figura 16: Cambio de tamaño de la ventana del entorno de simulación

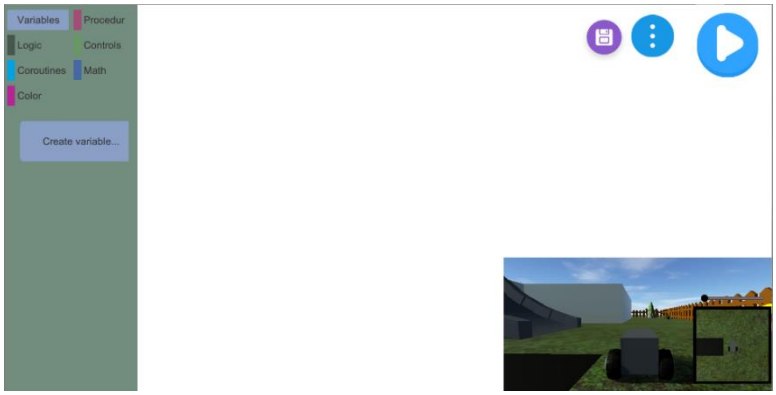

Figura 17: Escena de simulación con UBlockly integrado

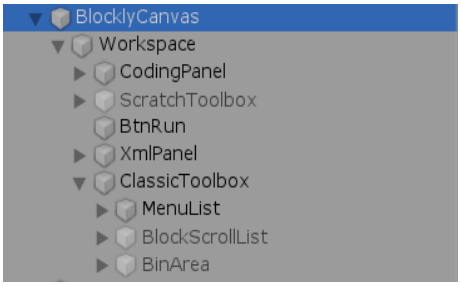

Figura 18: Estructura del Workspace

Como podemos apreciar en la **figura 17**, la zona de toolbox contiene varias categorías, estas son las que vienen por defecto si no hemos editado nada de BlockResSettings. Estas categorías cuentan con muchos bloques útiles, con los cuáles podremos crear bucles, condicionales, operadores de comparación, operadores matemáticos, etc.

El siguiente objetivo fue el de crear nuestras propias categorías. La primera que se añadió fue la de "MOVE". Esta categoría debía contar con las órdenes que el robot iba a desempeñar, como serían desplazarse hacia delante o hacia atrás y girar.

Para llevar a cabo esta tarea, en primer lugar, se tuvieron que crear tres nuevos ficheros en la carpeta **AllRes/Robot**. El primero que creamos fue "moveBlocks" que contiene la definición de los bloques de la categoría MOVE, el segundo fue "move_en", que contiene el diccionario con las palabras que usarían los mensajes de los bloques, y en tercer lugar "toolbox_robot", donde se definen las categorías y los colores que tendrán los bloques de estas, las categorías que contendrá por el momento este toolbox son las predefinidas más la categoría que vamos a añadir.

```
{
    "Style": "classic",
    "BlockCategoryList": [
         {
              "CategoryName": "VARIABLE",
              "ColorHex": "#8a9fc6",
              "BlockTypePrefix": "variables"
         },
        //Resto de categorías predeterminadas
          {
               "CategoryName": "MOVE",
               "ColorHex": "#ff0000",
               "BlockTypePrefix": "move"
          },
    ]
}
```

Figura 19: Fichero definición de toolbox (toolbox_robot)

```
{
    "type": "move_robot",
    "message0": "%{BKY_MOVE_ROBOT}
                %{BKY_MOVE_DISTANCE}
                %1",
    "args0": [
        {
            "type": "field_number",
            "name": "DISTANCE",
            "value": 0,
            "min": 0,
            "max": 10,
            "int": true
        }
    ],
    "previousStatement": null,
    "nextStatement": null
}
```

Figura 20: Fichero definición de bloque (moveBlocks)

```
{
    "MOVE": "Mover",
    "MOVE_ROBOT": "Mover robot",
    "MOVE_DISTANCE": "a velocidad"
}
```

Figura 21: Fichero diccionario (move_en)

En segundo lugar, se editó el fichero **BlockResSettings**, situado en la carpeta ProjectData. Para ello debemos buscarlo y hacer doble clic y nos saldrá una ventana en el inspector con muchos parámetros que podemos modificar. Aquí debemos arrastrar los tres ficheros que hemos creado ficheros a la zona que corresponda, en I18n Files añadimos el fichero de diccionario, en Block Json Files el fichero de definición de bloques y en ToolBox Files el fichero de definición de toolbox. Además, debemos añadir la ruta de dónde se guardarán los prefabs de los bloques, en nuestro caso, en **ProjectData/BlockGenPrefabs.**

Las **figuras 22**, **23** y **24** tendrían la configuración resultante de las modificaciones que hemos llevado a cabo en el fichero BlockResSettings:

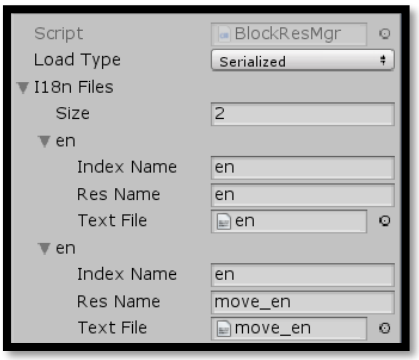

Figura 22: BlockResSettings diccionario

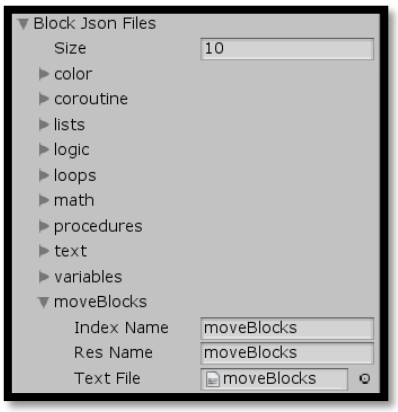

Figura 23: BlockResSettings bloques

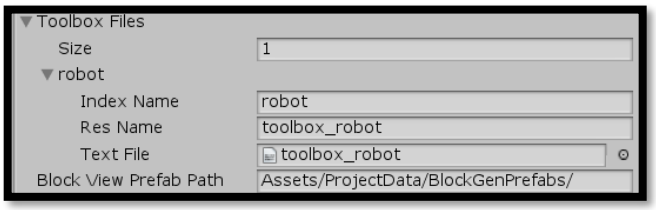

Figura 24: BlockResSettings toolbox

La configuración que hemos establecido dará como resultado al ejecutar la acción de construir los bloques el siguiente prefab de bloque de la categoría MOVE:

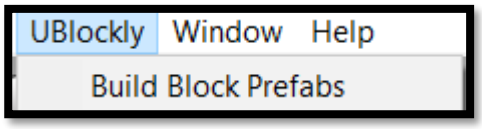

Figura 25: Construir bloques

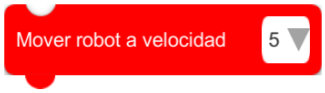

Figura 26: Bloque de movimiento

Hasta el momento había conseguido crear el prefab de un bloque, pero este al ser ejecutado no hace nada, ya que no tiene ningún fragmento de código asociado. Por lo que la siguiente tarea a realizar fue definir una acción para el bloque.

Para hacer esto me dirigí a la carpeta **Scripts/Robot** y creé dos ficheros con extensión C#, **RobotCmds** y **RobotController**.

En el archivo **RobotCmds** será donde asignaremos la función que hará un determinado bloque. Si hubiese algún parámetro en el bloque, se recogen los valores de esos campos para a continuación

pasarlos a otra función que estará en el fichero **RobotController**, que será donde se defina el comportamiento del bloque.

Las **figuras 27** y **28** nos muestran el código de una primera versión del asignación y definición de comportamiento del bloque de movimiento hacia delante.

```
public class Move_Robot_Cmdtor : EnumeratorCmdtor
    {
        protected override IEnumerator Execute(Block block)
        {   string distanceStr = block.GetFieldValue("DISTANCE");
            int distance = int.Parse(distanceStr);

            yield return RobotController.Instance.DoMove(distance);
        }
    }
```

Figura 27: Fichero de asignación de función (RobotCmds)

```
public IEnumerator DoMove(int distance){
    robot.GetComponent<Rigidbody>().AddForce(Vector3.forward * distance * 100);
    yield return null;
}
```

Figura 28: Fichero de definición de comportamiento (RobotController)

Esta primera versión no es la más óptima para ejecutar un movimiento controlado, ya que si observamos detenidamente el código de **RobotController**, estamos aplicando una fuerza de empuje que aunque se pueda modificar su intensidad, puede llevar a que sea impreciso. A pesar de todo, este paso supuso un antes y un después en el proyecto, ya que poder ejecutar el bloque correctamente nos indicó que todo estaba funcionando según lo esperado y al haber descubierto como crear este bloque no sería muy complicado crear otros siguiendo la misma metodología para el resto de movimientos que debía hacer el robot.

Por lo que me puse a trabajar para crear todos los bloques que fueran necesarios para la categoría de movimientos del robot y mejorar el sistema de avance del robot. Las acciones que se habían planeado para los movimientos del robot son las que enumeramos en la siguiente lista:

- Avance y retroceso con velocidad variable durante un periodo de tiempo.
- Avance y retroceso con velocidad variable con tiempo indefinido.
- Giro hacia la derecha o izquierda con grados ajustables.
- Giro hacia la derecha o izquierda indefinido.
- Detenención o parada.

Investigando cómo mejorar el sistema de avance, descubrí que las ruedas del robot tienen un componente llamado **HingeJoint**, que sirve para crear una unión entre las ruedas y la carrocería del robot, permitiendo que puedan girar y hacer que el robot se mueva. Dicho componente cuenta con un parámetro denominado “motor”, que permite hacer que la rueda gire en ambos sentidos y a distintas velocidades. Gracias a este descubrimiento rediseñé la función de avance del robot, sustituyendo la fuerza de empuje por un accionamiento de los motores. También añadí unas restricciones en la rotación que pueden tener las ruedas durante este avance, para procurar que sea lo más recto posible, además de hacer unos ajustes en el rozamiento de estas para que no hagan efecto de derrape al desarrollar el movimiento.

```
public IEnumerator DoMoveForward(int distance,int time){

            robot_rb.constraints = RigidbodyConstraints.FreezeRotation;
            wheel_r_rb.drag = 5; wheel_l_rb.drag = 5; wheel_c_rb.drag = 5;
             //Asignación de rozamiento y restringiendo rotación innecesaria de las ruedas
            JointMotor motor_r,motor_l,motor_c;
            motor_r = wheel_r.motor; motor_l = wheel_l.motor; motor_c = wheel_c.motor;

            motor_r.targetVelocity = 300*distance;
            motor_l.targetVelocity = -300*distance;
            motor_c.targetVelocity = -300*distance/3;

            motor_r.force = 200*distance;
            motor_l.force = 200*distance;
            motor_c.force = 200*distance/3;

            wheel_r.motor = motor_r;
            wheel_l.motor = motor_l;
            wheel_c.motor = motor_c;
            //Asignación de potencia y velocidad a las ruedas
            wheel_r.useMotor = true; wheel_l.useMotor = true; wheel_c.useMotor = true;
             //Accionando los motores
        if (time != 0) {//Condición para el bloque que tenga tiempo definido
                yield return new WaitForSeconds(time);//Para los motores al acabar el tiempo
                wheel_r.useMotor = false; wheel_l.useMotor = false; wheel_c.useMotor = false;
            }
            yield return null;
        }
}
```

Figura 29: Redefinición MoveForward (RobotController)

Este script sirve tanto para movimiento de avance indefinido como avance por un periodo de tiempo, ya que si es indefinido, se le pasa a esta función el parámetro time igual a 0. La función de retroceso del robot es exactamente igual que esta pero cambiando los signos de las velocidades de las ruedas.

En cuanto al desarrollo de las funciones para el giro de las ruedas, se aprovechó que cada rueda tenía un motor independiente, por ejemplo, si hacemos que la rueda derecha gire hacia delante y la izquierda hacia atrás conseguimos un giro del robot hacia la izquierda. Este script es bastante interesante, sobretodo las condiciones de parada del giro, que es donde centraremos la explicación.

```
...//Accionamiento de las ruedas para giro a la izquierda
      while (girando && angle != 12345){
        if (final_angle == robot.GetComponent<Transform>().eulerAngles.y) {
            girando = false;
        }
        if (final_angle <= epsilon || final_angle >= 360-epsilon){
            if (initial_angle > final_angle){ //Ángulo final cercano a 0 por la derecha
                if (robot.GetComponent<Transform>().eulerAngles.y - epsilon <= final_angle){
                    girando = false;
                }
            }
            else{ //Ángulo final cercano a 0 por la izquierda
                if (robot.GetComponent<Transform>().eulerAngles.y - epsilon >= final_angle){
                    girando = false;
                }
            }
        }
        if ((girando) & (robot.GetComponent<Transform>().eulerAngles.y < final_angle) &
           (robot.GetComponent<Transform>().eulerAngles.y <= initial_angle) & (!inverted))
        {}
        else{
            if ((girando) & (robot.GetComponent<Transform>().eulerAngles.y > final_angle)){
                inverted = true;
            }
            else{
```

```
                girando = false;
            }
        }
        yield return new WaitForEndOfFrame();
    }
    if (angle != 12345){ //Corrección del ángulo final
        Transform fix_turn = robot.GetComponent<Transform>();
        fix_turn.eulerAngles = new Vector3(fix_turn.eulerAngles.x,final_angle,fix_turn.eulerAngles.z);
    }
...//Detención de los motores de las ruedas
```

Figura 30: Condiciones de parada de giro a la izquierda (RobotController)

En concreto este fragmento de giro es para la condición de parada de giro a la izquierda, muy similar a la de giro a la derecha cambiando algunos símbolos en los condicionales.

Parte de este código está inspirado en un script que ya había sido desarrollado, aunque este no era del todo correcto, ya que bajo algunas circunstancias provocaba fallos con ángulos finales cercanos a 0º y no detenía el giro del robot, por lo que hubo que hacerle algunas modificaciones.

En este script hacemos varias comprobaciones para determinar si debemos detener el giro, en primer lugar y la condición más sencilla, si el ángulo actual es el ángulo objetivo, después comprueba si el ángulo final es cercano a 0 tanto por la izquierda como por la derecha, si el ángulo del robot llega a este umbral, también dejaría de girar. Tras esto hay una condición que verifica si el ángulo de giro del robot es menos que el final y el ángulo inicial y no está activado un booleano *inverted*, esta condición nos sirve para casos en los que el ángulo final es mayor que el inicial, si no se cumple comprueba que el ángulo del robot es mayor que el final, si no es así detiene el giro. Por último, realiza una corrección del ángulo de giro por si al detener el giro las ruedas se llegaran a mover por la inercia del movimiento.

Estas condiciones de parada solo se dan cuando se marca la cantidad de grados que se debe girar, en el caso de que no se haga tan solo se activará el giro sin frenar las ruedas.

Dicho esto, con este código conseguí crear los giros a ambos lados, tanto con grados ajustables como indefinidos, por lo que solo quedaría crear un bloque para detener el movimiento del robot. Este bloque tiene una implementación bastante simple, ya que lo que se hace es desactivar el motor y aumentar el rozamiento de las ruedas hasta un punto en el que no puedan seguir rodando.

```
    public IEnumerator DoStop(){
        robot_rb.constraints = RigidbodyConstraints.FreezeRotation;

        wheel_r.useMotor = false; wheel_r_rb.drag = 9999999;
        wheel_l.useMotor = false; wheel_l_rb.drag = 9999999;
        wheel_c.useMotor = false; wheel_c_rb.drag = 9999999;
        yield return new WaitForSeconds(1);
    }
```

Figura 31: Código de detención de movimiento

Al haber concluido la definición del comportamiento de los bloques, y por supuesto, haber creado la estructura de estos debidamente para crear los prefabs de estos, tal y como se hizo con el primer bloque de movimiento, el resultado obtenido sería el que podemos ver en la siguiente figura.

Figura 32: Repertorio de bloques de la categoría MOVE

Después de hacer varias pruebas para corroborar que los bloques funcionaban adecuadamente, llegó el turno de crear los bloques de los sensores, los cuáles nos permitirán condicionar las acciones del robot en función de la información que estos recojan del escenario. Para este TFG se han implementado bloques para los sensores de ultrasonido, contacto e infrarrojo, descartando el uso de los sensores laser y telémetro láser ya que el sensor ultrasonido puede sustituir a ambos.

Ya que los sensores tienen una función distinta al movimiento, se decidió crear una nueva categoría para almacenar estos bloques. El método para crear esta categoría es idéntico al que usamos para instanciar la categoría MOVE, creamos dos ficheros en **AllRes/Robot**, "sensorBlocks" y "sensor_en", que almacenarían la definición de los bloques y el diccionario respectivamente. También hubo que editar el fichero "robot_toolbox" para añadir la categoría SENSOR, además de añadir a **BlockResSettings** los dos nuevos ficheros que habíamos creado.

La creación de estos bloques tuvo algunas dificultades frente a los bloques de movimiento, la primera fue que los sensores debían estar etiquetados de alguna forma para hacer referencia a un sensor específico en el bloque, en segundo lugar, que los bloques debían retornan valores.

Para solucionar el primer problema decidí etiquetarlos cambiando su nombre al instanciarlos, de modo que se tuvieron que hacer unas modificaciones en las funciones Start() de los ficheros "**IRSensorScript**", "**USSensorScript**" y "**TouchSensorScript**", añadiendo un fragmento de código donde se buscan cuantos sensores hay del tipo que se pretende instanciar y se renombra el sensor con el tipo de este seguido del número del sensor que corresponde a la búsqueda realizada. He de destacar que el funcionamiento de la búsqueda es el mismo para todos los sensores.

```
  void Start () {
    …//Código de definición del sensor
    //Búsqueda de sensores en el robot
       int count = 0;
       foreach (Transform side in go.transform) {
           foreach (Transform tipoSensor in side){
               if(tipoSensor.tag == "tipo"){
                   count++;
                   Debug.Log(count);
                   if (GameObject.Find("SensorTipo"+count.ToString())==null){
                   //Si no hay un sensor con este nombre
                   //Se para la ejecución y se asigna dicho nombre
                      break;
       }}}}
       gameObject.name = "SensorTipo"+count.ToString();
}}
```

Figura 33: Renombrar sensores instanciados para su etiquetado

Para dar solución al segundo problema de retorno de valores, decidí revisar la definición de algunos bloques predefinidos que eran parte de la librería. Entre los que pude encontrar varios ejemplos que me sirvieron para comprender cómo retornar valores con un bloque.

```
[CodeInterpreter(BlockType = "sensor_us")]
public class Detection_Sensor_US_Cmdtor : ValueCmdtor{
    protected override DataStruct Execute(Block block){
        int index = int.Parse(block.GetFieldValue("NUMBER"));
        double data = RobotController.Instance.INFOSensorUS(index);
        Debug.Log("Devuelve "+data.ToString());
        return new DataStruct(data); //Retorna el valor de la función
}}
```

Figura 34: Función de asignación sensor ultrasonido (RobotCmds)

```
public double INFOSensorUS(int index){
    double distance;
    USSensorDistance us = (USSensorDistance)
                          GameObject.Find("SensorUS"+index).transform.GetChild(16)
                          .GetComponent(typeof(USSensorDistance));
    if (us.getDetection ()) {
        distance = (double)us.getDistanceHit ();
    } else
        distance = double.MaxValue;
    return distance;
}
```

Figura 35: Función de definición sensor ultrasonido (RobotController)

A diferencia de las funciones de movimiento, la función de **RobotCmds** recoge el valor que retorna la función de **RobotController** de tipo *double***,** que toma la información del sensor, para que el bloque devuelva dicho valor. Por último, la función "*Detection_Sensor_US_Cmdtor*" retornaría una estructura de datos que contiene el valor que nos ofrece el sensor. El resto de las funciones de sensores son similares a las mostradas en las dos figuras anteriores.

Después de seguir los pasos correspondientes que detallamos anteriormente en este documento para definir y crear el comportamiento de los bloques, obtenemos el siguiente conjunto de bloques para la categoría de los sensores, para dar como resultado la toolbox completa con todos los bloques necesarios para el control del robot.

Figura 36: Repertorio de bloques de la categoría

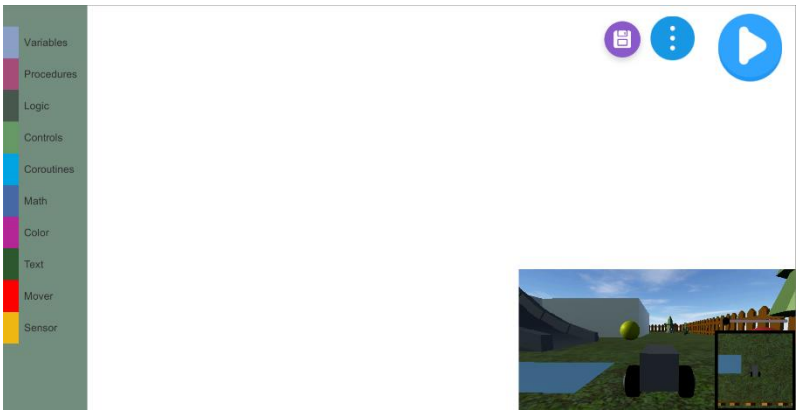

Figura 37: Escena de simulación con las categorías MOVE y SENSOR

Haber conseguido finalizar con éxito la creación de esta última categoría daría por concluida la parte de integración de los tres proyectos.

3.3.3 Mejoras de funcionalidades e interfaz

En este apartado detallaremos todas las mejoras que se han hecho tanto a aspectos visuales como funcionales para mejorar la experiencia de usuario y hacer la interfaz más cómoda y sencilla.

Vista de las etiquetas de sensores

Una de estas mejoras fue mostrar las etiquetas de los sensores al usuario, para que este pudiera distinguir los sensores entre sí. Hubo que hacer modificaciones en los prefabs de los sensores añadiendo un texto que sería el que mostraría la etiqueta y escribir una línea en los ficheros de creación de sensores (**figura 33**), en la que se iguala el texto de la etiqueta al nombre del sensor.

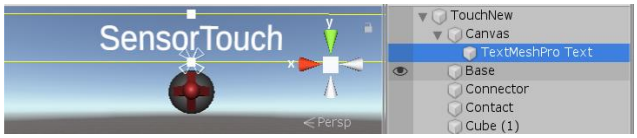

Figura 38: Añadiendo etiqueta de sensor

```
public TextMeshProUGUI sensorname;
...
gameObject.name = "SensorTipo"+count.ToString();
sensorname.text = "SensorTipo"+count.ToString();
```

Figura 39: Modificando texto de la etiqueta

Además, también se llevó a cabo una configuración de modo que si estamos creando o editando un sensor, su etiqueta cambia de color para indicarnos sobre cuál estamos trabajando. Para lograrlo se tuvo que añadir un par de líneas que cambiaran el color del sensor, tanto el el script de creación **CrearSensor** como en el script que permite editarlos, llamado **OptionsSensorFront**.

```
//CrearSensor
    public void crearTouchBack() {
        GameObject touchGO = (GameObject)Instantiate (touchPrefab,placeBack.position,placeBack.rotation,parentBack);
        options.setGo (touchGO);
        touch = (TouchSensorScript) touchGO.GetComponent (typeof(TouchSensorScript));
        options.setUSScript (us);
        //Añadir las siguientes dos líneas a las funciones de creación de sensores Touch, IR y US
        text = GameObject.Find(touch.name + "/Canvas/TextMeshPro Text").GetComponent<TextMeshProUGUI>();
        text.color = new Color(255,255,0,255);
        ...
    }
//OptionsSensorFront
    void Update () {
      ...
      if (go.tag == "US" || go.tag == "Touch" || go.tag == "Laser" || go.tag == "IR" || go.tag == "Lidar"){
        text = GameObject.Find(go.name + "/Canvas/TextMeshPro Text").GetComponent<TextMeshProUGUI>();
        text.color = new Color(255,255,0,255);
      }
    }
    public void setSlidersLaserToGameObject(){
        ...
        if (go.tag == "US" || go.tag == "Touch" || go.tag == "IR"){
            text = GameObject.Find(go.name + "/Canvas/TextMeshPro Text").GetComponent<TextMeshProUGUI>();
            text.color = new Color(255,255,255,255);
        }
        go = null;
    }
```

Figura 40: Cambio de color de las etiquetas de los sensores

La primera función, localizada en el fichero **CrearSensor** serviría para cambiar el color al crear el sensor, en este caso sería para el de contacto trasero, por lo que habría que añadir esta modificación para las demás funciones de creación. En la función "Update" del archivo **OptionsSensorFront** cambiamos el color a las etiquetas de los sensores que estamos tratando de modificar, y en la tercera y última función, restablecemos el color de la etiqueta al original.

Por último, para que las etiquetas solo sean visibles en el menú de creación, esta es desactivada al pasar a la escena de simulación. Esto es posible gracias a los scripts **Rotation.cs** y **RobotCollider.cs**, ya que como expliqué en el punto 3.3.1, la activación y desactivación de estos permite cambiar parámetros y aspectos del robot dependiendo de la escena en la que nos encontremos. La activación

de las etiquetas debería ir en la función “OnEnable” de **Rotation**, al ser el script que está habilitado en la escena de creación, el método que nos permite ocultar los nombres de los sensores debe estar ubicada en la misma función, pero en el fichero **RobotCollider** que es habilitado en la simulación.

```
GameObject sensor_id;
for (int i=2; i<7; i++){
   for (int j=2; j< gameObject.transform.GetChild(i).childCount; j++){
      sensor_id = gameObject.transform.GetChild(i).
                  GetChild(j).GetChild(0).gameObject;
      sensor_id.SetActive(false);
      //Para habilitar cambiar SetActive a true
   }
}
```

Figura 41: Desactivación de etiquetas (RobotCollider)

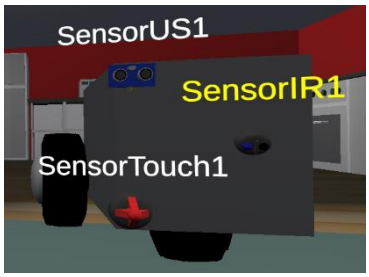

Figura 42: Etiquetas de sensores en el robot

Creación de esquema del robot en entorno de simulación

Para hacer más sencilla la tarea de recordar cómo era el diseño de nuestro robot, se pensó que sería una buena idea agregar una copia de este en la escena de simulación que sirviera de guía para consultar las etiquetas de los distintos sensores con los que cuenta nuestra creación. Para hacerlo, al cambiar a la escena de simulación se crea una copia del robot al que se le hacen algunas modificaciones, concretamente tiene los ajustes del robot de la escena de creación, de modo que puede girar si se pulsan las teclas de las flechas en el teclado, además de poder ver las etiquetas de los sensores. Esto se ha conseguido habilitando el script **Rotation** y desactivando **RobotCollider**, además de deshabilitar **RobotScriptController** para que no se hagan comprobaciones de en qué escena está este GameObject y altere los scripts que este puede usar.

```
if (SceneManager.GetActiveScene().name == "Simulator"){
     if (GameObject.Find("Robot_Esquema") == null){
        var temp = Instantiate(gameObject);
        temp.name  = "Robot_Esquema";
        rscriptCTLRsq = temp.GetComponent<RobotScriptsController>();
        rotationCSsq = temp.GetComponent<Rotation>();
        rcolliderCSsq = temp.GetComponent<RobotCollider>();
        rscriptCTLRsq.enabled = false;
        rotationCSsq.enabled = true;
        rcolliderCSsq.enabled = false;
     }
}
```

Figura 43: Creación y configuración de scripts de Robot_Esquema (RobotScriptController)

Por último, en la función “Start” de **Rotation** se ajusta la posición y rotación del robot, además de eliminar cualquier restricción que pueda tener el robot para rotar.

```
    void Start(){
        if (gameObject.name == "Robot_Esquema"){
            robot_tr = gameObject.GetComponent<Transform>();
            robot_tr.localScale = robot_tr.localScale * 6;
            robot_tr.localPosition = new Vector3(10005.5f,10001,10054);
            robot_tr.localEulerAngles = new Vector3(0,180,0);
            robot_cp = gameObject.GetComponent<Rigidbody>();
            robot_cp.constraints = RigidbodyConstraints.None;
    }}
```

Figura 44: Cambio de posición, rotación y restricciones de Robot_Esquema (Rotation)

Con estas modificaciones se consiguió crear la copia del robot en la escena de simulación, este sería el resultado.

Figura 45: Esquema del robot en la escena de simulación

También se modificó la fuente de las etiquetas de los sensores para que fuera más legible, optando por amarillo como color normal, y verde como color de sensor seleccionado en la escena de creación, además de acentuar los bordes de las letras.

Cambio de tamaño de la ventana del escenario de simulación

Pensamos que algunos usuarios querrían ver el entorno de simulación en pantalla completa para poder apreciar mejor el escenario y ver el recorrido del robot con más claridad. Por este motivo se añadió un botón que permitiera alternar el tamaño de la ventana en tiempo de ejecución.

Hubo un pequeño problema para poder llevar esto a cabo, y es que al colocar el botón en la escena de simulación y querer buscar la cámara en el robot, al pertenecer a tipos de escenas distintos no la encontraba ya que la cámara era hija de "*DontDestroyOnLoad*" al ser un objeto que no se destruye con el paso entre escenas, y el botón de cambio de tamaño de "*Simulador*", por eso se añadió este botón a Permanente, de modo que así no ocasionaría problemas.

```
public void MinOrMax(){
        if (cam_.rect == new Rect(0.65f,-0.45f,1,0.8f)){
            //Se hace grande
            cam_.rect = new Rect(0,0,1,1);
        }
        else{
            //Se hace pequeño
            cam_.rect = new Rect(0.65f,-0.45f,1,0.8f);
        }
    }
```

Figura 46: Cambio de tamaño de la ventana del entorno de simulación

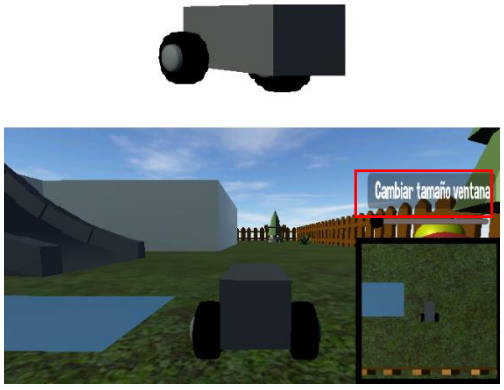

Figura 47: Botón de cambio de tamaño de ventana

Al pulsar el botón de cambio de tamaño, se llama a la función "MinOrMax" del fichero **Maximize_On_Click.cs**. Lo que se hace en este método es alternar el parámetro "rect" de la cámara "Robot Camera" que tomará un valor u otro dependiendo del valor actual de este.

Reiniciar escena

Esta nueva funcionalidad dará al usuario la oportunidad reiniciar la escena de simulación, de modo que, si se ha equivocado con las instrucciones que ha encomendado al robot, al pulsar el botón de reinicio, se detendrá la ejecución del código del Workspace y se llevará el robot hasta el punto de inicio para que lo pueda volver a intentar.

Después de algo de búsqueda por el código fuente de la librería uBlockly, descubrí que existe una función que permite detener la ejecución del intérprete de código C# mientras se conservan los bloques que se han puesto en el Workspace, de modo que el usuario no pierde el progreso que ha hecho con la construcción de su script.

En la **Figura 48** podemos observar la función del nuevo fichero **ResetTransform.cs** que se ejecuta cuando queremos reiniciar la escena.

```
public void ResetPosRot(){
    robot_tr.position = new Vector3(131.4f,8.5f,-76.1f);
    robot_tr.eulerAngles = new Vector3(0,0,0);

    CSharp.Interpreter.Stop();//Detención del intérprete
    //Deteniendo robot
    robot_rb.constraints = RigidbodyConstraints.FreezeRotation |
    RigidbodyConstraints.FreezePositionX |
     RigidbodyConstraints.FreezePositionZ;

    wheel_r.useMotor = false; wheel_r_rb.drag = 9999999;
    wheel_l.useMotor = false; wheel_l_rb.drag = 9999999;
    wheel_c.useMotor = false; wheel_c_rb.drag = 9999999;
}
```

Figura 48: Función de reinicio de la simulación

Figura 49: Botón de reinicio de simulación

Además de parar la interpretación de bloques y devolver el robot a la posición de origen, también se apagan los motores de las ruedas y se aumenta la fricción de estas con el suelo, junto con restricciones de movimiento salvo en el eje Y, de modo que cualquier movimiento que estuviera haciendo el robot es completamente detenido.

Vuelta a la escena de creación

Nos planteamos que el usuario por accidente podría equivocarse con el diseño de sensores, por lo que se decidió incluir un botón que permitiera volver a la escena de creación, para que pueda corregir los problemas que tenga su robot. Por consiguiente, decidimos habilitar un botón en la escena de simulación que llevara a cabo esta función.

En la **figura 50** podemos ver que para cambiar de escena hemos añadido unas líneas al fichero **MainMenu.cs** que, además, detienen la interpretación de código C# si se estuviera ejecutando algún bloque en la escena de simulación. Este método se ejecuta al pulsar el botón "*Volver a creación*"

```
public void CreateRobotSq () {
    CSharp.Interpreter.Stop();
    SceneManager.LoadScene ("CreationSq");
}
```

Figura 50: Regreso a la escena de creación y parada del intérprete

Figura 51: Botón de retorno a la escena de creación

Existieron algunos problemas a la hora de implementar esta funcionalidad, ya que los scripts para realizar la personalización de sensores del robot no almacenaban las referencias a los GameObjects de los distintos lados del robot al haberse referenciado por el inspector del editor de Unity, esto provocaba que al volver a la escena de creación no se pudiera editar el robot. La solución consistió en hacer la referencia directamente en el código, concretamente en la función "Start" del fichero **CrearSensor.cs**. Podremos ver un fragmento de este código en la **figura 52** en el que asignamos los GameObjects de la parte delantera a sus variables en el código, debe hacerse lo mismo para el resto de las partes del robot (*Back, Left, Right y Top*)**.**

```
    void Start () {
        ...
        parentFront = GameObject.Find("Permanente/Robot/FrontSide").GetComponent<Transform>();
        placeFront = GameObject.Find("Permanente/Robot/FrontSide/FrontTransform").GetComponent<Transform>();
        placeFrontIR = GameObject.Find("Permanente/Robot/FrontSide/FrontTransformIR").GetComponent<Transform>();
        ...
    }
```

Figura 52: Fragmento de código de asignación de lados del robot al fichero CrearSensor.cs

Esta complicación también ocurría al intentar cambiar el tamaño de las ruedas y del chasis del robot, aunque en este punto se planteó que esta funcionalidad no tenía tanta importancia como se había planteado el pasado curso académico, por lo que decidimos eliminar los botones que permitían editar estos aspectos del modo de creación, contando únicamente con la funcionalidad de editar sensores.

Además de esto, también se tuvo que definir algunos parámetros en las funciones OnEnable de los ficheros **Rotation** y **RobotCollider** para cambiar los parámetros de robot no solo una vez (que es lo que nos permite la función "Start") sino aplicar estos cambios cada vez que se cambia de escena y se habilitan estos scripts. Entre estos cambios destacan el de la cámara "Robot Camera" que debe volver a ocupar la totalidad de la pantalla al cambiar a la escena de creación.

Ocultar bloques de sensores no presentes en el diseño del robot

Para evitar fallos y confusiones a los usuarios de la aplicación a la hora de crear los scripts con los bloques, decidimos ocultar aquellos bloques relacionados con sensores que no ha añadido a su robot, ya que no necesitará usarlo.

Examinando detenidamente la librería UBlockly encontramos la función en la que se instancian los bloques en las distintas categorías. Modificamos esta función de modo que se comprueba el bloque que se pretende instanciar, si este bloque es de la categoría SENSOR se comprueba que el robot cuenta con algún sensor que esté relacionado con el bloque que estamos evaluando, y solo en el caso de que lo encuentre, incluye el bloque en la vista de la categoría.

```
protected virtual void BuildBlockViewsForActiveCategory()
        {
            Transform contentTrans = mRootList[mActiveCategory].transform;
            var blockTypes = mConfig.GetBlockCategory(mActiveCategory).BlockList;
            //Mostrar solo bloques de los sensores con los que cuenta el robot
            foreach (string blockType in blockTypes)
            {
                switch(blockType){
                    case "sensor_ir_white":
                        if (GameObject.FindObjectOfType<IRSensorDetection>() != null){
                            NewBlockView(blockType, contentTrans);
                        }
                        break;
                    case "sensor_ir_black":
                        if (GameObject.FindObjectOfType<IRSensorDetection>() != null){
```

```
                    NewBlockView(blockType, contentTrans);
                }
                break;
            case "sensor_touch_contact":
                if (GameObject.FindObjectOfType<TouchSensorContact>() != null){
                    NewBlockView(blockType, contentTrans);
                }
                break;
            case "sensor_touch_notcontact":
                if (GameObject.FindObjectOfType<TouchSensorContact>() != null){
                    NewBlockView(blockType, contentTrans);
                }
                break;
            case "sensor_us":
                if (GameObject.FindObjectOfType<USSensorDistance>() != null){
                    NewBlockView(blockType, contentTrans);
                }
                break;
            default:
                NewBlockView(blockType, contentTrans);
                break;
        }
    }
    //Mostrar solo bloques de los sensores con los que cuenta el robot
}
```

Figura 53: Modificación para ocultar bloques de sensores (ClassicToolbox.cs)

En la **figura 54** podemos ver el resultado que hemos obtenido al realizar esas modificaciones al código de instanciación de bloques de nuestra toolbox.

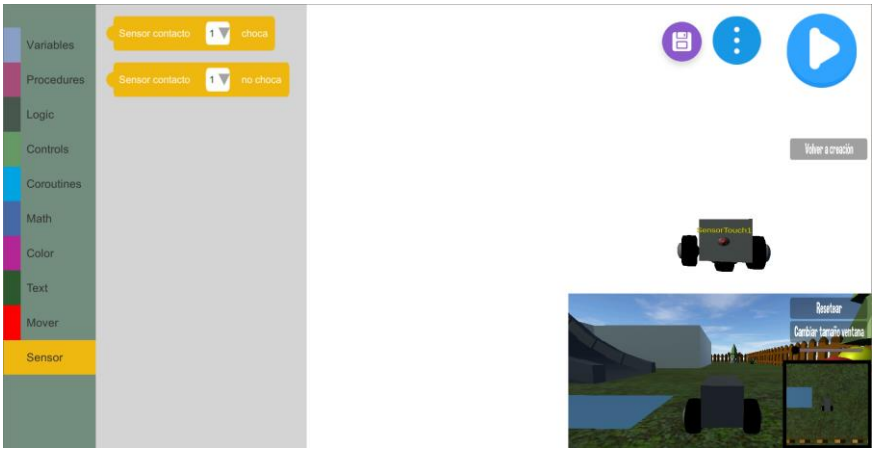

Figura 54: Categoría SENSOR con los bloques relacionados con los sensores del robot

Aumentar y disminuir el tamaño de los bloques de área de trabajo

A pesar de poder arrastrar el cursor para poder la totalidad de nuestro script visual, hemos notado que el área de trabajo podría ser algo incómoda para trabajar en el caso de que tengamos muchos bloques en ella, por lo que incluimos unos botones en la esquina inferior izquierda por medio de los cuáles podremos aumentar y disminuir el tamaño de los bloques que se encuentran en el área de trabajo.

Se escribió un pequeño script (**ModCodingPanelSize.cs**) con dos funciones, las cuáles son llamadas al pulsar los botones para aumentar y disminuir la escala del área de trabajo. Podemos ver estas funciones en la **figura 55.**

```
public void MakeitBigger(){
    if(codingpaneltr.localScale.x < 1.33){
        codingpaneltr.localScale += new Vector3(0.11f,0.11f,0);
    }
}
public void MakeitSmaller(){
    if(codingpaneltr.localScale.x > 0.66){
        codingpaneltr.localScale -= new Vector3(0.11f,0.11f,0);
    }
}
```

Figura 55: Cambiar escala del panel de codificación

Figura 56: Botones para cambiar escala del panel de codificación

3.3.4 Creación de retos

Una vez se llevaron a cabo la integración de todos los módulos en uno solo y se realizaron mejoras en la funcionalidad e interfaz de la aplicación, llegamos a la fase de desarrollo en la que creamos los retos, con los cuáles podremos poner a prueba las habilidades de los usuarios y la capacidad del robot de solucionar situaciones por medio de las instrucciones que le proporcionemos y el uso de sus sensores.

Se diseñaron un total de dos retos para el robot, uno para el sensor de infrarrojo, que consiste en seguir dentro de una línea blanca para llegar a su objetivo, y otro para los sensores de contacto y ultrasonido, en la que el robot tendrá que recorrer un tramo laberíntico hasta capturar una moneda. Cada uno de estos retos es una escena distinta, ya que el escenario ha sido modificado para cada una de estas pruebas, aunque el resto de los elementos son exactamente iguales.

Para poder resolver las pruebas, los usuarios deberán tener información sobre cómo es el terreno del entorno de simulación.

Se han preparado dos botones en la escena de simulación que nos permiten ir a cada uno de estos retos.

Reto del sensor infrarrojo

El sensor infrarrojo permite evaluar si una zona del terreno por la que está pasando el robot es blanca o es negra. Basándonos en las capacidades de este sensor, se ideó una prueba que consistiría en un camino blanco sobre un fondo negro que tiene al final una moneda. Para recoger la moneda se deberá aprovechar la información que reciba el sensor para seguir la ruta y superar el reto.

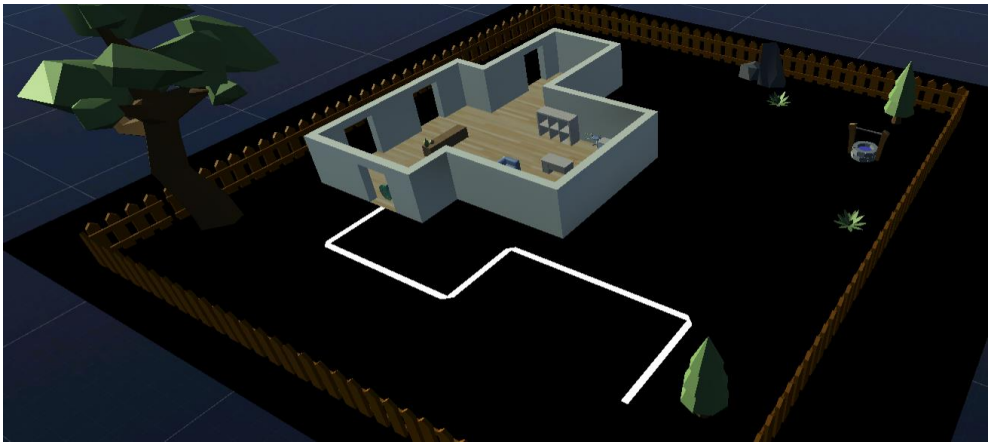

Figura 57: Entorno de simulación: Reto para Sensor Infrarrojo

Como podemos ver en la **figura 57** el reto cuenta con un camino que tiene tanto rectas como curvas, por lo que se el usuario deberá hacer buen uso de los sensores para superar esta prueba.

Para crear este reto hay que arrastrar un “Material” de color negro al GameObject “Plane” para pintar el plano de este color. El camino fue creado a base de otros pequeños GameObjects del mismo tipo que “Plane” de color blanco, a los que se le modificaron la escala, posición y rotación para unirlos y formar un camino. Las curvas fueron hechas simplemente creando otro objecto “Plane” de color negro y superponiéndolo en las esquinas como se puede apreciar en la **figura 58**.

Figura 58: Plane negro superpuesto

Una posible solución a este reto consistiría, en primer lugar, en la creación de un robot que cuente con dos sensores infrarrojos con precisión máxima y alcance rondando los 0.3 metros, situados en la parte delantera, posicionados uno en cada extremo lo más cercanos al suelo posible, tal y como se muestra en la **figura 59**.

Figura 59: Configuración de robot: Reto para Sensor Infrarrojo

En segundo lugar, se debería elaborar a base de la unión de distintos bloques el script de la **figura 60.**

Figura 60: Script para superar prueba: Reto para Sensor Infrarrojo

Como podemos ver a simple vista, es un script bastante sencillo, aunque usamos algunos elementos bastante interesantes, como son los bucles y los condicionales. Básicamente el robot podrá hacer una de las cuatro acciones que podemos ver en la lista del siguiente párrafo, las cuales están condicionadas por lo que detecten los sensores infrarrojos.

1. Si ambos sensores reciben “blanco”, el robot **avanza**.
2. Si el sensor situado a la derecha recibe “negro” y el izquierdo detecta “blanco”, el robot realizará un **giro a la izquierda**.
3. Si el sensor situado a la izquierda recibe “negro” y el derecho detecta “blanco”, el robot realizará un **giro a la derecha**.
4. Si ambos sensores reciben “negro”, el robot **retrocede**.

Reto del sensor de contacto y sensor ultrasonido

El sensor de contacto envía una señal en el caso de que detecte una colisión con otro objeto, el ultrasonido capta la distancia a la que capta un objeto en un rango de 2.5 metros. Con esto en mente, barajamos la posibilidad de hacer un reto que pudiera solucionarse usando alguno de estos dos sensores, ya que, si lo pensamos bien, las capacidades de ambos son relativamente similares ya que ambos detectan la ubicación de un cuerpo, uno por medio del contacto y otro por medio de un haz de luz.

Se ideó un reto que permita explotar las posibilidades de ambos sensores, en el que el objetivo será recorrer un camino laberíntico hasta el final y recoger una moneda. El laberinto consiste en la unión de varios rectángulos alargados, los cuáles son inamovibles.

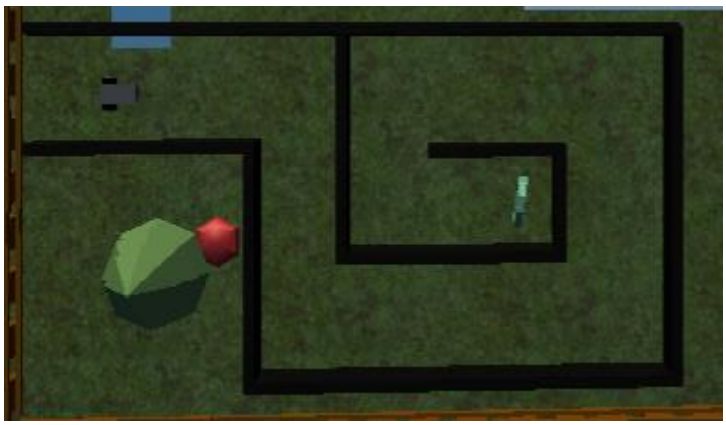

Figura 61: Entorno de simulación (Vista aérea): Reto para Sensor de Contacto y Ultrasonido

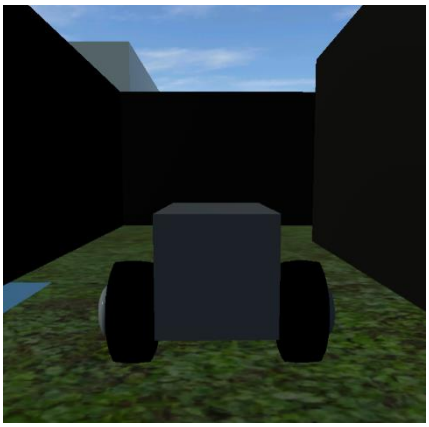

Figura 62: Entorno de simulación (Vista robot): Reto para Sensor de Contacto y Ultrasonido

Tal y como vemos en la **figura 61** el camino cuenta con varias curvas y paredes que dificultarán recorrer el camino. Para llegar hasta el final del recorrido, podemos crear algunos de los robots que se mostrarán en las **figuras 63** y **64.**

Figura 63: Configuración de robot: Reto para Sensor Ultrasonido

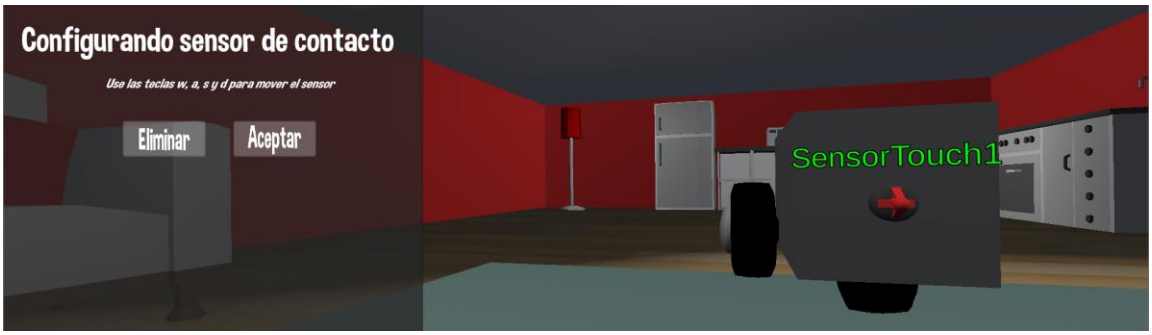

Figura 64: Configuración de robot: Reto para Sensor de contacto

Ambos sensores deben estar colocados en el frontal del robot, de modo que detecten los objetos por delante. El rango del sensor ultrasonido lo hemos establecido a 2.5 metros para que tenga bastante anticipación a la colisión con las paredes del laberinto.

En las **figuras 65** y **66** tenemos unos ejemplos de scripts que conseguirían hacer que el robot recorra el camino de forma satisfactoria.

Figura 65: Script para superar prueba: Reto para Sensor de contacto

Figura 66: Script para superar prueba: Reto para Sensor Ultrasonido

Estos scripts son bastante similares en cuanto a las condiciones que tienen, ya que tienen que realizar el mismo recorrido, girando a la derecha 90º grados una vez y girando a la izquierda 90º cinco veces. Hay algunas variaciones, como que por ejemplo el sensor de contacto retrocede un poco al chocar con la pared, para ganar algo de espacio para el giro y el script de sensor ultrasonido no cuenta con esta acción ya que el robot estará lo suficientemente alejado de la pared como para poder girar correctamente. Otra de las diferencias que existen son las condiciones para ejecutar esas acciones, ya que mientras que el robot con el sensor de contacto debe avanzar si no hay choque, el robot del sensor ultrasonido avanza hasta que la distancia entre él y la pared es menor que tres metros.

La curiosidad de este reto es que a pesar de ser el mismo, debemos pensar de forma distinta al afrontarlo con un sensor u otro, ya que la dificultad radica en entender cómo se comportan y comprender que devuelven distinta información.

3.3.5 Exportar proyecto a HTML5

Después de haber finalizado el desarrollo del proyecto, el último paso era compilar el proyecto para poder lanzarlo a navegadores. Esta tarea es bastante sencilla, ya que simplemente nos hemos tenido que dirigir a la barra de herramientas del editor de Unity, pulsar en *File->Build Settings* (**figura 67)**. Al hacer esto vemos que se nos abre una ventana, con varias plataformas a la que exportar, seleccionamos HTML5, desactivamos todas las casillas de configuración de la exportación y pulsamos *Build And Run* y seleccionar la carpeta donde se guardarán los ficheros generados (**figura 68**).

Una vez pasados unos quince minutos ya estará exportado, nos dirigimos a la carpeta donde se han guardado estos ficheros y arrastramos el fichero index.html al navegador (en mi caso he usado Mozilla Firefox) y podemos comprobar que se lanza correctamente (**figura 69**).

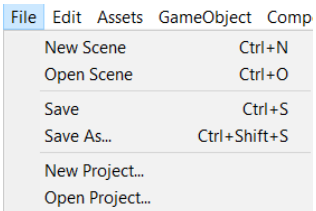

Figura 67: File->Build Settings en el editor de Unity

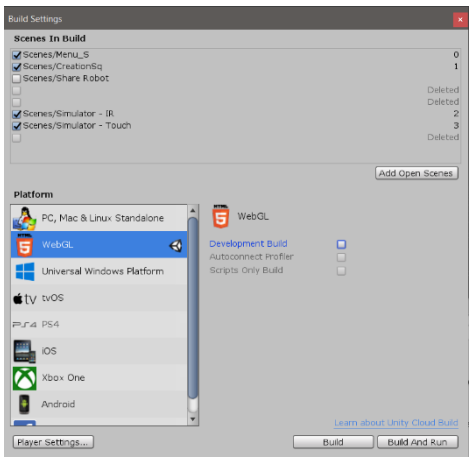

Figura 68: Ventana de exportación de proyecto

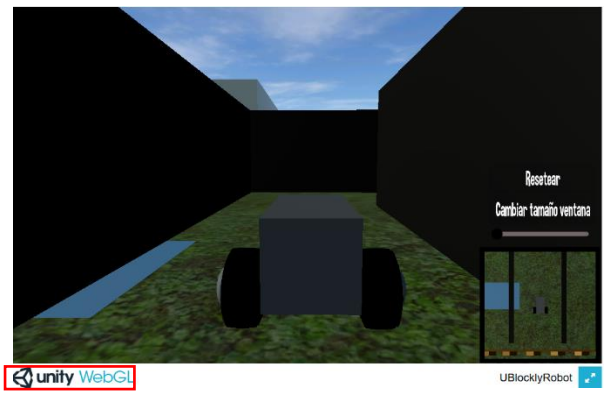

Figura 69: Proyecto en el navegador

Capítulo 4
Caso de uso

Al iniciar la aplicación lo primero que veremos será este pequeño menú (**figura 68**).

Figura 70: Menú de inicio

Debemos pulsar sobre la opción “Crear un robot” para pasar a la escena de creación. Una vez aquí, debemos personalizar nuestro robot para poder superar alguno de los dos retos propuestos. En este caso, vamos a orientar este caso de uso a la resolución del reto de sensor de ultrasonido.

Si queremos añadir el sensor de ultrasonido, debemos dirigirnos a la parte izquierda, en la que tenemos las opciones para añadir sensores y pulsar sobre el botón “Ultrasonido”. Una vez hayamos pulsado este botón, se añadirán algunas opciones a la pantalla, las cuáles nos permiten elegir la parte del robot en la cual colocaremos el sensor. En este caso, vamos a pulsar sobre la opción “Parte frontal” (**figura 71**).

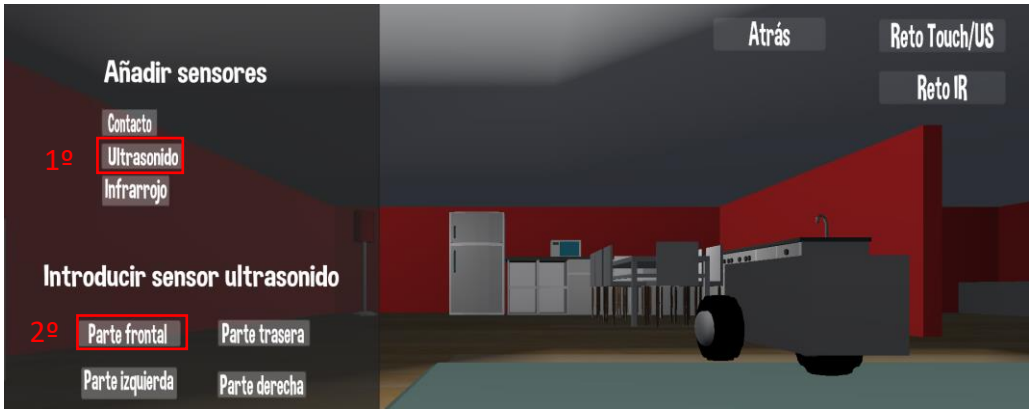

Figura 71: Añadiendo un sensor de ultrasonido a la parte delantera del robot.

Cuando pulsemos ese botón se nos desplegará un menú con distintas opciones que podemos personalizar de este sensor, en este caso podemos alterar el rango de detección de objetos del sensor y la precisión de aciertos que va a tener. Si arrastramos las barras situadas a la izquierda, podremos alterar estos parámetros. Para cambiar la posición del sensor pulsaremos las teclas W,A,S y D. Nosotros vamos a situar el sensor más o menos en el centro de la parte frontal con la configuración de precisión y rango que vemos en la **figura 72.** Cuando hayamos terminado de configurar este sensor, pulsamos en el botón “Aceptar” para regresar al menú de personalización.

Figura 72: Personalizando sensor ultrasonido

De vuelta en el menú de creación, pulsaremos en el botón "Reto Touch/US" para ir al entorno de simulación con la prueba correspondiente. En el caso de que hayamos cometido algún error en la configuración del sensor ultrasonido, podemos tanto modificarlo como eliminarlo si pulsamos con nuestro cursor encima de él.

Figura 73: Preparativos para iniciar el reto de sensor ultrasonido

Al pulsar en este botón se cargará la escena de simulación junto con el área de trabajo sobre la cuál crearemos nuestro script visual. Para comenzar a programar debemos pulsar sobre alguna de las categorías de la toolbox, momento en el que se desplegarán los bloques de esa categoría. Acto seguido debemos elegir alguno de esos bloques y arrastrarlo hasta el área de trabajo.

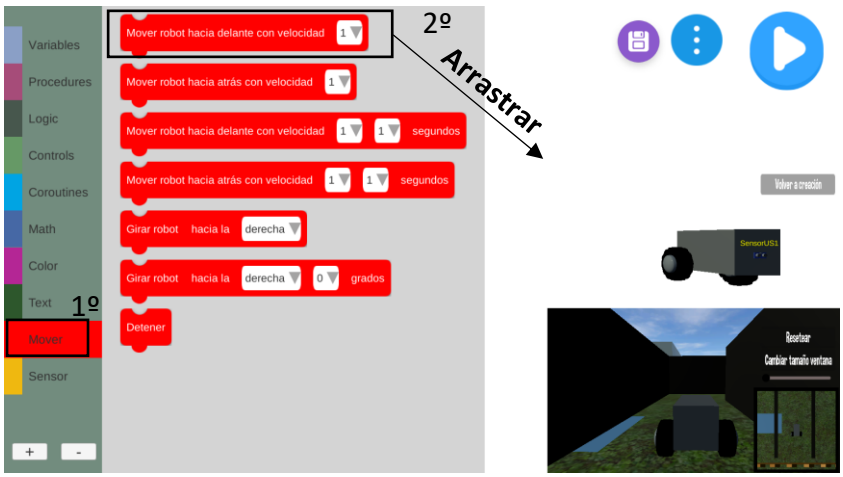

Figura 74: Llevar un bloque al área de trabajo

Una vez tengamos este bloque en el área de trabajo, vamos a tener que encadenar distintos bloques para formar el script de la **figura 66.** Encadenar bloques es tan sencillo como arrastrar un bloque al área de trabajo y acercarlo al bloque al que queremos unirlo, cuando estén lo suficiente cerca, la ranura por la que se va a encajar el bloque cambiará de color. Si en este momento soltamos el bloque que estamos arrastrando, los dos bloques quedarán unidos.

Figura 75: Encajar dos bloques

En algunos bloques vamos a tener campos de selección o de introducción numérica. Cuando pulsemos sobre estos campos nos saldrá un cuadro de diálogo. Con estos cuadros de diálogo podemos especificar la velocidad del robot, los ángulos de giro, la dirección del giro, el sensor del que recibimos la información y muchas otras cosas.

Figura 76: Cuadro de diálogo: Introducción de número

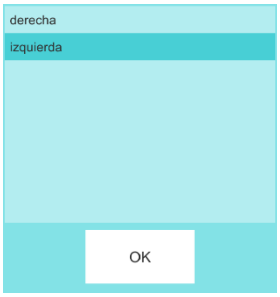

Figura 77: Cuadro de diálogo: Selección de una opción

Después de encadenar bloques conseguiremos el script que resolverá el reto. Para ejecutar el script debemos pulsar en el botón situado en la parte superior derecha. Una vez haya dado comienzo la simulación podemos observar tanto la traza de ejecución del código por medio de un punto verde, que se irá posicionando encima del bloque que está ejecutando el intérprete, además de comprobar cómo va avanzando el robot por el escenario. Si en algún punto de la ejecución queremos reiniciar la escena porque algo ha ido mal, simplemente debemos pulsar sobre el botón "Resetear" para volver al punto de partida.

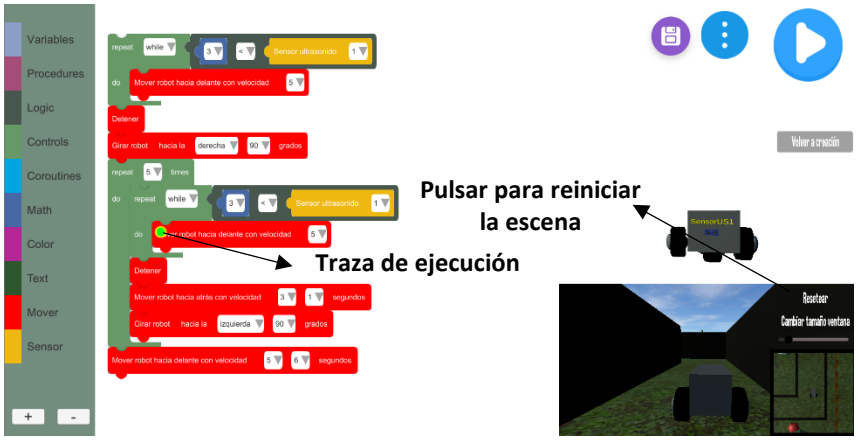

Figura 78: Ejecutando bloques del área de trabajo

Si conseguimos que la ejecución haya sido satisfactoria y queremos volver a la escena de creación, solo debemos pulsar sobre el botón "Volver a creación" y cambiar el diseño del robot.

Capítulo 5
Conclusiones

A modo de conclusión, podemos dar por cumplidos los objetivos que se habían planteado para este trabajo de fin de grado, ya que no solo se han conseguido integrar de forma más sólida los tres módulos, sino que se han integrado todos usando únicamente la tecnología Unity.

El hecho de que todos los módulos estén integrados dentro de una misma aplicación hace que su uso sea más fácil y cómodo para el usuario, ya que, si lo comparamos con la versión anterior de esta línea de trabajo, debíamos cargar un total de tres veces ficheros de configuración para aplicaciones distintas. Ahora todo está condensado en una única aplicación, la cual se ha conseguido exportar a HTML5, lo que nos permitirá que cualquiera pueda usar esta aplicación fácilmente al alojarla en un servidor web.

Además de la mejora de usabilidad, se han añadido muchas mejoras tanto a nivel de funcionalidad como a nivel visual, por ejemplo, poder ampliar la pantalla del simulador, cambiar el tamaño de los bloques o poder reiniciar la simulación son valores añadidos que enriquecen la experiencia de uso.

No nos debemos olvidar de que también se han añadido algunos retos, ya que, sin estos, la aplicación queda algo vacía ya que no podemos poner a prueba los robots que creamos. Es cierto que tan solo hay dos retos, pero son suficientes para hacer una demostración de que todos los módulos han sido debidamente integrados y que todo funciona.

Este proyecto sienta las bases de una aplicación de robótica educacional que tiene mucho potencial de crecimiento, por lo que en futuras líneas de trabajo se podrían añadir más retos, además de catalogarlos por orden de dificultad, e incluso crear un sistema por el que los usuarios puedan hacer sus propios retos de forma fácil. Teniendo esto en mente, sería un gran valor añadido que la aplicación formara parte de una comunidad educativa online, de modo que se pudieran subir y compartir los retos que se creen entre todos los usuarios, además de añadir algún sistema que permita puntuar y hacer rankings según la destreza de los usuarios a la hora de resolver las pruebas.

Otras mejoras que se podrían desarrollar en fututos trabajo serían añadir otros tipos de chasis al robot que permita una distribución distinta de las ruedas, ya que esto podría dar bastante juego a la hora de plantear los retos, o incluso añadir partes móviles, por medio de las cuáles el autómata pueda recoger elementos del escenario y transportarlos.

Capítulo 6
Summary and Conclusions

In conclusion, we can say that the objectives for this end-of-degree work has been accomplished, since not only have the three modules been more robustly integrated, as well have been integrated using Unity technology.

The fact that all modules are integrated within the same application makes their use easier and more comfortable for the user, because, if we compared it with the previous version of this line of work, we should load a total of three times configuration files in different applications. Now everything is condensed into a single application that was able to be exported to HTML5, which will allow anyone to use it easily if we host it on a web server.

In addition to the improved usability, many improvements have been added both at the functionality level and at the visual level, for example, being able to expand the simulator screen, resize the blocks or be able to restart the simulation are elements that enrich the experience of use.

We must not forget that some challenges have been added, because without them, the application could look empty for the fact that we couldn´t test the robots that we create. It is true that there are only two challenges, but they are enough to demonstrate that all modules have been properly integrated and that everything it's working.

This project lays the foundation for an educational robotics application that has a lot of growth potential, so in future lines of work more challenges could be added, as well as classify them in order of difficulty, and even create a system by which users can easily make their own challenges. With this in mind, it would be great input if the app were part of an online educational community, so that challenges that are created among all users could be uploaded and shared, as well as adding some system that allows to score and do rankings according to the prowess of users in solving tests.

Other improvements that could be developed in future work would be to add other types of chassis to the robot allowing for a different distribution of wheels, since this could give a lot of possibilities when it comes to setting the challenges, or even adding moving parts, through which the automaton can pick up elements from the stage and transport them.

Capítulo 7
Presupuesto

En la **tabla 1** podemos ver el prepuesto correspondiente al trabajo realizado en este proyecto.

7.1 Presupuesto

Tarea	Precio por hora	Horas empleadas	Coste
Investigación acerca de las tecnologías empleadas en el proyecto y familiarización con el mismo	14€	35h	490€
Integración del módulo de creación y simulación	18€	40h	720€
Integración de UBlockly en el proyecto	25€	70h	1750€
Funcionalidades adicionales	18€	40h	720€
Creación de retos	18€	15h	270€
Testeo del proyecto	20€	60h	1200€
Elaboración de la memoria	14€	50h	700€
TOTAL		310h	**5850€**

Tabla 1**:** Tabla de presupuesto

Bibliografía

[1] Robótica educativa – Wikipedia
https://es.wikipedia.org/wiki/Rob%C3%B3tica_educativa

[2] ¿Qué es la robótica educativa y por qué nos gusta tanto?
https://www.juguetronica.com/blog/que-es-la-robotica-educativa-y-por-que-nos-gusta-tanto/

[3] Pensamiento computacional – Wikipedia
https://es.wikipedia.org/wiki/Pensamiento_computacional

[4] Beneficios del pensamiento computacional
https://codelearn.es/beneficios-del-pensamiento-computacional/

[5] Blockly | Google Developers
https://developers.google.com/blockly/

[6] Qué es la robótica educativa | Nuevo sistema de enseñanza
https://edukative.es/que-es-la-robotica-educativa/

[7] B. Ortega-Ruipérez and M. A. Mikel, "Robótica DIY: pensamiento computacional para mejorar la resolución de problemas," vol. 17, no. 2, pp. 129–143, 2018.

[8] Capítulo 2: Abstracción
http://www.pensamientocomputacional.org/Files/02Capitulo.pdf

[9] Coding Robots Curriculum Outline – CoderZ
https://gocoderz.com/coding-robots/

[10] Educators – CoderZ
https://gocoderz.com/educators/

[11] Vrep: Simulación de robots virtuales – robologs
https://robologs.net/2016/01/22/vrep-simulacion-de-robots-virtuales/

[12] Coppelia Robotics V-REP: Create. Compose. Simulate. Any Robot: Features
http://www.coppeliarobotics.com/features.html

[13] Robot Dash de Wonder Workshop - Apple (ES)
https://www.apple.com/es/shop/product/HJYC2VC/A/robot-dash-de-wonder-workshop

[14] Unity (Motor de Juego) – Wikipedia
https://es.wikipedia.org/wiki/Unity_(motor_de_juego)

[15] Products – Unity
https://unity3d.com/es/unity

[16] ¿Sabes que es UNITY? Descúbrelo aquí – De Idea a App
https://deideaaapp.org/sabes-que-es-unity-descubrelo-aqui/

[17] Creando y usando scripts - Unity Manual
https://docs.unity3d.com/es/current/Manual/CreatingAndUsingScripts.html

[18] Rigidbody - Unity Manual
https://docs.unity3d.com/es/current/Manual/class-Rigidbody.html

[19] Prefabs – Unity Manual
https://docs.unity3d.com/es/current/Manual/Prefabs.html

[20] Escenas - Unity Manual
https://docs.unity3d.com/es/current/Manual/CreatingScenes.html

[21] GitHub imagicbell/UBlockly: Reimplementation of Google Blockly for Unity
https://github.com/imagicbell/UBlockly

[22] Google Blockly Reimplementation With Unity/C#(1) Magicbell's Blog
http://magicbell.beanstu.io/unity/2017/10/11/blockly-one.html
[23] Google Blockly Reimplementation With Unity/C#(2) Magicbell's Blog
http://magicbell.beanstu.io/unity/2017/10/14/blockly-two.html
[24] Google Blockly Reimplementation With Unity/C#(3) Magicbell's Blog
http://magicbell.beanstu.io/unity/2017/10/22/blockly-three.html
[25] Google Blockly Reimplementation With Unity/C#(4) Magicbell's Blog
http://magicbell.beanstu.io/unity/2017/10/31/blockly-four.html
[26] Microsoft Word – Wikipedia
https://es.wikipedia.org/wiki/Microsoft_Word
[27] ¿Qué es Microsoft Word y para qué sirve?
https://quees.guru/para-que-sirve-el-programa-microsoft-word/